New Data Protection Abstractions for Emerging Mobile and Big Data Workloads

Riley Spahn

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2020

ABSTRACT

New Data Protection Abstractions for Emerging Mobile and Big Data Workloads

Riley Spahn

Two recent shifts in computing are challenging the effectiveness of traditional approaches to data protection.Emerging machine learning workloads have complex access patterns and unique leakage characteristics that are not well supported by existing protection approaches. Second, mobile operating systems do not provide sufficient support for fine grained data protection tools forcing users to rely on individual applications to correctly manage and protect data. My thesis is that these emerging workloads have unique characteristics that we can leverage to build new, more effective data protection abstractions.

This dissertation presents two new data protection systems for machine learning workloads and a new system for fine grained data management and protection on mobile devices. First is Sage, a differentially private machine learning platform addressing the two primary challenges of differential privacy: running out of budget and the privacy utility tradeoff. The second system, Pyramid, is the first selective data system. Pyramid leverages count featurization to reduce the amount of data exposed while training classification models by two orders of magnitude. The final system, Pebbles, provides users with logical data objects as a new fine grained data management and protection primitive allowing data management at a higher level of abstraction. Pebbles, leverages high level storage abstractions in mobile operating systems to discover user recognizable application level data objects in unmodified mobile applications.

Contents

List of Figures

List of Tables

Acknowledgements

None of my research nor my thesis would have been possible without help from many people. First I would like to thank my advisors Roxana Geambasu and Gail Kaiser. They've provided invaluable guidance and support during my time at Columbia. I'll also be eternally grateful to Dave Evans for welcoming me into his lab at Virginia. I'd also like to thank Siddhartha Sen for the opportunity to collaborate with Microsoft Research and Jason Nieh for serving on my committee.

I'm very fortunate to have had the opportunity to collaborate and work with countless wonderful colleagues. In particular, it's been a privilege to collaborate with Mathias Lécuyer for better part of six years. In no particular order I would also like to thank Jon Bell, Fang-Hsiang Su, Michael Z Lee, Ian King, Sravan Bhamidipati, Anthony Narin, Giannis Spiliopoulos, Kiran Vodrahalli, Augustin Chaintreau, Daniel Hsu, and Vaggelis Atlidakis.

My research was supported in part by a three-year Google Ph.D. Fellowship, NSF CNS-1351089, CNS-1514437, CNS-0905246, CCF-1161079, CCF-1302269, NIH U54 CA121852, DARPA FA8650-11-C-7190, Google Faculty Research Awards, a Google Cloud grant, a Microsoft Faculty Fellowship, a Sloan Faculty Fellowship, and Schmidt Futures.

Finally, I would like to thank my parents Bev and Jim, and my sister Ryan for always encouraging me to keep going. Most importantly, I need to thank my spouse Kate LeCroy and dog Magnolia. Kate supported and encouraged me every step of the way and through every crazy deadline. Magnolia provided exceptional emotional support by sitting under my chair each day.

To my family

Chapter 1

Introduction

The past decade has seen two major shifts in computing away from the traditional desktop and server workloads. First is the shift to machine learning and big data workloads. These are an increasingly common and intensive workload class[7] that are driving research at all layers of the compute stack[89;9]. Second is the shift to mobile computing which long ago overtook PCs as the most common mode of personal computing[148]. Both of these shifts are challenging existing data protection abstractions.

Enabled by shrinking compute costs, and driven by the perceived value of big data, companies and organizations have embraced the promise of machine learning. Machine learning and big data have led to advancements in diverse academic research areas like medicine[78] and physics[20], and proven to be valuable for commercialization through areas such as ad targeting[108] and machine translation[177]. This perception of data's untapped value has driven aggressive personal data collection. Companies pool collected data in wide access data lakes[5;38] with loose access policies to enable the same data to be used across diverse models and use cases. Those models trained on the data in the common pool are deployed to data centers and other devices around the world. This approach to data collection, management, and use leads to two exposure risks.

The first exposure risk is that data may leak through the trained models themselves. Many of the most effective commercial models are deep neural networks with the capacity to encode and effectively memorize significant amounts of information about the training data. It's

possible and even practical to extract sensitive encoded information about the training data directly from the models themselves[154;30]. Despite the fact that these models encode information about the training data and should be treated as sensitive assets, models are routinely deployed to untrusted and hostile domains like mobile devices[159;6] for the purpose of issuing low latency predictions for applications such as predictive typing[133]. Even if only deployed to secure systems in data centers, those models may be extracted and stolen through prediction APIs[166]. This cavalier behavior does not treat trained models as the sensitive assets they are.

The second exposure risk is the possibility of the raw data itself being compromised by malicious hackers infiltrating a system or by unscrupulous employees snooping on data for their own reasons[73]. Recent news has shown numerous examples of such incidents. Hundreds of millions of records were stolen as part of a breaches at the Marriott hotel company[142] and the Equifax credit reporting agency[24]. These breaches are dwarfed by a breach at Yahoo that compromised an astounding three billion records[69]. Securing large data stores is difficult because data access patterns under machine learning workloads differ from those under traditional workloads. Training a machine learning model requires repeated access to full datasets during the entire model training and retraining cycle. Such repeated access makes common security procedures such as tight access control and fine grained auditing nearly impossible. This increasing the risk of data exposure and makes forensic investigation in the case of a breach more difficult.

Traditional approaches to protection such as access control and encryption are necessary, but insufficient, for machine learning and big data workloads. Even if we were able to create very specific access control lists so that each model can only access the data that it requires, data will be useful for many different applications and models, and the access control lists will eventually degrade into wide access. For example, user location data may be useful for fraud detection in a payment processing application or for ad targeting and recommendation

based on travel patterns. Encryption, like access control lists, are useful but sufficient for these emerging workloads. Using encrypted computation it's possible to train models without ever exposing plaintext data[75]. The ciphertext data may be stolen but it will be of little use to any adversary largely mitigating the second exposure risk. Unfortunately this approach is only theoretical because encrypted computation is not yet practical for computationally intense workloads. Even if encryption and access control were practical to apply, they do nothing to avert the first exposure risk. Even with no plaintext exposure, adversaries can still exfiltrate data through the models themselves. These traditional approaches to protection, while necessary, are insufficient for the complexities and requirements of machine learning and big data workloads.

The second shift challenging existing protection approaches is mobile computing. Mobile devices offer increased productivity through pervasive connectivity allowing ubiquitous access information and applications. The new mobile computing form factor brings with it, new high level paradigms for users to interact with the system. For the first time, users primarily interact with applications rather than the operating system. These new interaction paradigms separate users from the trusted operating system and the traditional data management abstractions. Instead of exposing traditional data management and access control abstractions through the file system or developing data management abstractions appropriate for the new paradigm, mobile operating systems have entirely ceded data management responsibilities to individual applications. Each application must develop and expose a set of data management tools unique to that application and users must trust that the applications will faithfully execute their data management requests. Applications have shown time and time again that they cannot be trusted with the responsibility of managing user data. Despite the trust that users are forced to place in applications, there are near daily reports of intentional[169] and unintentional[61] user data mismanagement.

Providing meaningful data protection and management tools on mobile devices is not as

straight forward as simply exposing a set of tools through the file system as is Unix based operating systems. Mobile operating systems have evolved to provide much higher level storage abstractions that are not captured by the file system structure. Rather than just providing a file system, mobile operating systems provide abstractions such as relational databases and key value stores in addition to traditional files. The traditional file, directory, and access control list paradigm will not provide protection at a meaningful level of granularity. Any new data management and protection abstractions will need to be aware of and align with modern storage abstractions.

My thesis develops new data protection abstractions and systems that both support the needs of emerging ML and mobile workloads and are effective at lowering the data exposure risks created by them. My hypothesis is that we can leverage characteristics of these workloads to develop both practical and effective protection systems for them.

My thesis describes two new protection solutions for machine learning systems. Chapter 2 presents Sage. Sage addresses the first risk to machine learning systems using differential privacy[53]. Sage leverages data accumulated in a growing database to alleviate one of the foundational issues of differential privacy, running out of budget. Sage also presents a novel block composition technique to enable a new privacy-adaptive training approach to help navigate the privacy utility tradeoff. Chapter 3 describes Pyramid. Pyramid addresses the second exposure risk using existing featurization techniques. Machine learning models are commonly trained using features that are precomputed from historical data[86]. Pyramid retrofits one specific technique, count featurization[155], into a protection mechanism. Using count featurization, Pyramid reduces the amount of data exposed when training classification models by two orders of magnitude.

Finally, my thesis presents a system for data protection and management on mobile devices. Chapter 4 presents Pebbles. Pebbles is a fine-grained data management system that allows users to manage their data with a new and powerful abstraction, the application level

data object (LDO). Pebbles leverages structure inherent in modern storage abstractions to accurately recognize user level LDOs in application such as emails and documents in unmodified applications. Pebbles exposes LDOs through an API which we used to build several new and useful protection tools in a fine grained manner where they would otherwise need to rely on applications for data management and protection.

Chapter 2

Sage: Accounting and Quality Control in a Differentially Private Machine Learning Platform

2.1 Introduction

Machine learning (ML) is changing the origin and makeup of the code driving many of our applications, services, and devices. Traditional code is written by programmers and encodes algorithms that express business logic, plus a bit of configuration data. We keep sensitive data – such as passwords, keys, and user data – out of our code, because we often ship this code to untrusted locations, such as end-user devices and app stores. We mediate accesses to sensitive data with access control. When we do include secrets in code, or when our code is responsible for leaking user information to an unauthorized entity (e.g., through an incorrect access control decision), it is considered a major vulnerability.

With ML, "code" – in the form of ML models – is learned by an ML platform from *a lot of training data*. Learning code enables large-scale personalization, as well as powerful new applications like autonomous cars. Often, the data used for learning comes from users and is of personal nature, including emails, searches, website visits, heartbeats, and driving behavior. And although ML "code" is derived from sensitive data, it is often handled as secret-free code. ML platforms, such as Google's Tensorflow-Extended (TFX), routinely push models trained over sensitive data to servers all around the world[22;85;102;133] and sometimes to end-user devices[176;159] for faster predictions. Some companies also push feature models – such as user embedding vectors and statistics of user activity – into model stores that are often times widely accessible within the companies[85;102;153]. Such exposure would be inconceivable in a traditional application. Think of a word processor: it might push *your* documents to *your* device for faster access, but it would be outrageous if it pushed *your* documents to *my* (and everyone else's) device!

There is perhaps a sense that because ML models aggregate data from multiple users, they "obfuscate" individuals' data and warrant weaker protection than the data itself. However, this perception is succumbing to growing evidence that ML models can leak specifics about individual entries in their training sets. Language models trained over users' emails for

auto-complete have been shown to encode not only commonly used phrases but also social security numbers and credit card numbers that users may include in their communications[29]. Prediction APIs have been shown to be enable testing for membership of a particular user or entry within a training set[154]. Finally, it has long been established both theoretically and empirically that access to too many linear statistics from a dataset – as an adversary might have due to periodic releases of models, which often incorporate statistics used for featurization – is *fundamentally non-private*[19;46;60;87].

As companies continue to disseminate many versions of models into untrusted domains, controlling the risk of data exposure becomes critical. We present *Sage*, an ML platform based on TFX that uses differential privacy (DP)[51] to bound the cumulative exposure of individual entries in a company's sensitive data streams through all the models released from those streams. At a high level, DP randomizes a computation over a dataset (e.g. training one model) to bound the leakage of individual entries in the dataset through the output of the computation (the model). Each new DP computation increases the bound over data leakage, and can be seen as consuming part of a *privacy budget* that should not be exceeded; Sage makes the process that generates all models and statistics preserve a *global DP guarantee*.

Sage builds upon the rich literature on DP ML *algorithms* (e.g.,[10;182;106], see §2.2) and contributes pragmatic solutions two of the most pressing *systems challenges* of global DP: (1) running out of privacy budget and (2) the privacy-utility tradeoff. Sage expects to be given training pipelines explicitly programmed to individually satisfy a parameterized DP guarantee. It acts as a new access control layer in TFX that: mediates all accesses to the data by these DP training pipelines; instantiates their DP parameters at runtime; and accounts for the cumulative privacy loss from all pipelines to enforce the global DP guarantee against the stream. At the same time, Sage provides the developers with: control over the quality of the models produced by the DP training pipelines (thereby addressing challenge (2)); and the ability to release models endlessly without running out of privacy budget for the stream

(thereby addressing challenge (1)).

The key to addressing both challenges is the realization that ML workloads operate on *growing databases*, a model of interaction that has been explored very little in DP, and only with a purely theoretical and far from practical approach[45]. Most DP literature, largely focused on *individual algorithms*, assumes either static databases (which do not incorporate new data) or online streaming (where computations do not revisit old data). In contrast, *ML workloads* – which consist of many algorithms invoked periodically – operate on *growing databases*. Across invocations of different training algorithms, the workload both incorporates new data and reuses old data, often times adaptively. It is in that *adaptive reuse of old data coupled with new data* that our design of Sage finds the opportunity to address the preceding two challenges in ways that are practical and integrate well with TFX-like platforms.

To address the running out of privacy budget challenge, we develop *block composition*, the first privacy accounting method that both allows efficient training on growing databases and avoids running out of privacy budget as long as the database grows fast enough. Block composition splits the data stream into *blocks*, for example by time (e.g., one day's worth of data goes into one block) or by users (e.g., each user's data goes into one block), depending on the unit of protection (event- or user-level privacy). Block composition lets training pipelines combine available blocks into larger datasets to train models effectively, but accounts for the privacy loss of releasing a model at the level of the specific blocks used to train that model. When the privacy loss for a given block reaches a pre-configured ceiling, the block is *retired* and will not be used again. However, new blocks from the stream arrive with zero privacy loss and can be used to train future models. Thus, as long as the database adds new blocks fast enough relative to the rate at which models arrive, Sage will never run out of privacy budget for the stream. Finally, block composition allows adaptivity in the choice of training computation, privacy parameters, and blocks to execute on, thereby modeling the most comprehensive form of adaptivity in DP literature.

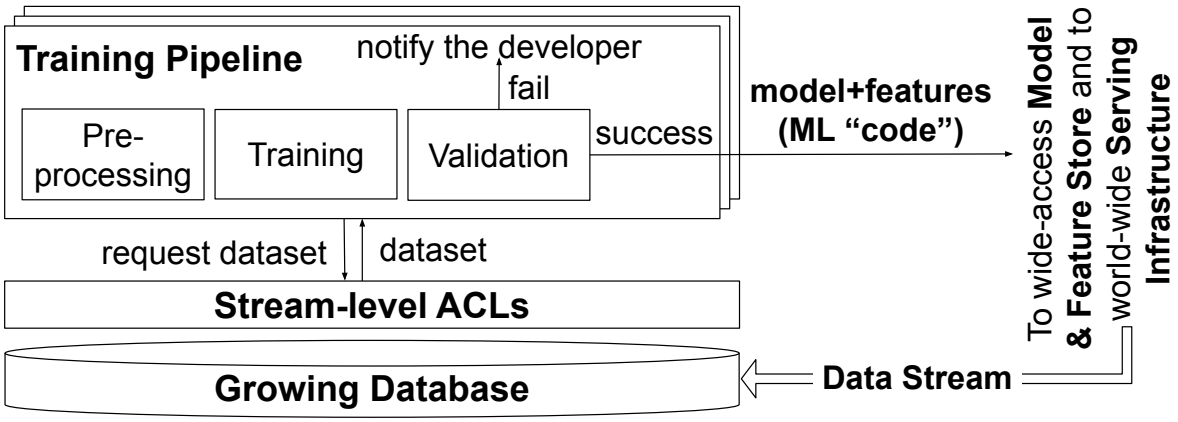


Figure 2.1: Typical Architecture of an ML Platform.

To address the privacy-utility tradeoff we develop *privacy-adaptive training*, a training procedure that controls the utility of DP-trained models by repeatedly and adaptively training them on growing data and/or DP parameters available from the stream. Models retrain until, with high probability, they meet programmer-specified quality criteria (e.g. an accuracy target). Privacy-adaptive training uses block composition's support for adaptivity and integrates well with TFX's design, which includes a model validation stage in training pipelines.

2.2 Motivation and Goals

Our effort builds upon an opportunity we observe in today's companies: the rise of *ML platforms*, trusted infrastructures that provide key services for ML workloads in production, plus strong library support for their development. They can be thought of as *operating systems* for ML workloads. Google has TensorFlow-Extended (TFX)[22]; Facebook has FBLearner[85]; Uber has Michelangelo[102]; and Twitter has DeepBird[99]. The opportunity is to *incorporate DP into these platforms as a new type of access control that constrains data leakage through the models a company disseminates*.

ML Platforms

Figure 2.1 shows our view of an ML platform; it is based on[22;85;102]. The platform has several components: *Training Pipelines* (one for each model pushed into production), *Serving Infrastructure*, and a shared data store, which we call the *Growing Database* because it accumulates data from the company's various streams. The access control policies on the Growing Database are exercised through Stream-level ACLs and are typically restrictive for sensitive streams.

The *Training Pipeline* trains a model on data from the Growing Database and verifies that it meets specific quality criteria before it is deployed for serving or shared with other teams. It is launched periodically (e.g., daily) on datasets containing samples from a representative time window (e.g., logs over the past month). It has three customizable modules: (1) *Pre-processing* loads the dataset from the Growing Database, transforms it into a format suitable for training and inference by applying feature transformation operators, and splits the transformed dataset into a training set and a testing set; (2) *Training* trains the model on a training set; and (3) *Validation* evaluates one or more *quality metrics* – such as accuracy for classification or mean squared error (MSE) for regression – on the testing set. It checks that the metrics reach specific *quality targets* to warrant the model's push into serving. The targets can be fixed by developers or can be values achieved by a previous model. If the model meets all quality criteria, it is bundled with its feature transformation operators (a.k.a. *features*) and pushed into the Serving Infrastructure. The model+features bundle is what we call *ML code*.

The *Serving Infrastructure* manages the online aspects of the model. It distributes the model+features to inference servers around the world and to end-user devices and continuously evaluates and partially updates it on new data. The model+features bundle is also often pushed into a company-wide *Model and Feature Store*, from where other teams within the company can discover it and integrate into their own models. Twitter and Uber report

sharing embedding models[153] and tens of thousands of summary statistics[102] across teams through their Feature Stores. To enable such wide sharing, companies sometimes enforce more permissive access control policies on the Model and Feature Store than on the raw data.

Threat Model

We are concerned with the increase in sensitive data exposure that is caused by applying looser access controls to data-carrying ML code –models+features– than are typically applied to the data. This includes placing models+features in company-wide Model and Feature Stores, where they can be accessed by developers not authorized to access the raw data. It includes pushing models+features, or their predictions, to end-user devices and prediction servers that could be compromised by hackers or oppressive governments. And it includes opening the models+features to the world through prediction APIs that can leak training data if queried sufficiently[166;154]. Our goal is to "neutralize" the wider exposure of ML codes by making the process of generating them DP across all models+features ever released from a sensitive stream.

We assume the following components are trusted and implemented correctly: Growing Database; Stream-level ACLs; the ML platform code running a Training Pipeline. We also *trust the developer* that instantiates the modules in each pipeline *as long as the developer is authorized* by Stream-level ACLs to access the data stream(s) used by the pipeline. However, we do not trust the wide-access Model and Feature Store or the locations to which the serving infrastructure disseminates the model+features or their predictions. Once a model/feature is pushed to those components, it is considered *released* to the untrusted domain and accessible to adversaries.

We focus on two classes of attacks against models and statistics (see Dwork[59]): (1) *membership inference*, in which the adversary infers whether a particular entry is in the training set based on either white-box or black-box access to the model, features, and/or

predictions[19;60;87;154]; and (2) *reconstruction attacks*, in which the adversary infers unknown sensitive attributes about entries in the training set based on similar white-box or black-box access[29;46;59].

Differential Privacy

DP is concerned with whether the output of a computation over a dataset – such as training an ML model – can reveal information about individual entries in the dataset. To prevent such information leakage, *randomness* is introduced into the computation to hide details of individual entries.

Definition 1 (Differential Privacy (DP)[57])**.** A randomized algorithm $\mathcal{Q} : \mathcal{D} \rightarrow \mathcal{V}$ is (ϵ, δ)-DP if for any $\mathcal{D}, \mathcal{D}'$ with $|D \oplus D'| \leq 1$ and for any $\mathcal{S} \subseteq \mathcal{V}$, we have: $P(\mathcal{Q}(\mathcal{D}) \in \mathcal{S}) \leq e^{\epsilon} P(\mathcal{Q}(\mathcal{D}') \in \mathcal{S}) + \delta$.

The $\epsilon > 0$ and $\delta \in [0, 1]$ parameters quantify the strength of the privacy guarantee: small values imply that one draw from such an algorithm's output gives little information about whether it ran on D or D'. The *privacy budget* ϵ upper bounds an (ϵ, δ)-DP computation's privacy loss with probability (1-δ). $\oplus$ is a dataset distance (e.g. the symmetric difference[111]). If $|D \oplus D'| \leq 1$, D and D' are *neighboring datasets*.

Multiple mechanisms exist to make a computation DP. They add noise to the computation scaled by its sensitivity s, the maximum change in the computation's output when run on any two neighboring datasets. Adding noise from a Laplace distribution with mean zero and scale $\frac{s}{\epsilon}$ (denoted $laplace(0, \frac{s}{\epsilon})$) gives $(\epsilon, 0)$-DP. Adding noise from a Gaussian distribution scaled by $\frac{s}{\epsilon}\sqrt{2\ln(\frac{1.25}{\delta})}$ gives (ϵ, δ)-DP.

DP is known to address the attacks in our threat model[154;59;29;88]. At a high level, membership and reconstruction attacks work by finding data points that make the observed model more likely: if those points were in the training set, the likelihood of the observed output

increases. DP prevents these attacks, as no specific data point can drastically increase the likelihood of the model outputted by the training procedure.

DP literature is very rich and mature, including in ML. DP versions exist for almost every popular ML algorithm, including: stochastic gradient descent (SGD)[10;182]; various regressions[34;122;161;184;94]; collaborative filtering[110]; language models[109]; feature selection[35]; model selection[156]; evaluation[26]; and statistics, e.g. contingency tables[21], histograms[180]. The privacy module in TensorFlow v2 implements several SGD-based algorithms[106].

A key strength of DP is its *composition* property, which in its basic form, states that the process of running an (ϵ_1, δ_1)-DP and an (ϵ_2, δ_2)-DP computation on the same dataset is $(\epsilon_1 + \epsilon_2, \delta_1 + \delta_2)$-DP. Composition enables the development of complex DP computations – such as DP Training Pipelines – from piecemeal DP components, such as DP ML algorithms. Composition also lets one account for (and bound) the privacy loss resulting from a sequence of DP-computed outputs, such as the release of multiple models+features.

A distinction exists between *user-level* and *event-level* privacy. User-level privacy enforces DP on all data points contributed by a user toward a computation. Event-level privacy enforces DP on individual data points (e.g., individual clicks). User-level privacy is more meaningful than event-level privacy, but much more challenging to sustain on streams. Although Sage's design can in theory be applied to user-level privacy (§2.3), we focus most of the paper on *event-level privacy*, which we deem practical enough to be deployed in big companies. §2.5 discusses the limitations of this semantic.

2.3 The Sage Architecture and Design

The Sage training platform enforces a global (ϵ_g, δ_g)-DP semantic over all models+features that have been, or will ever be, released from each sensitive data stream. The highlighted portions in Figure 2.2 show the changes Sage brings to a typical ML platform. First, each Training Pipeline must be made to individually satisfy (ϵ, δ)-DP for some privacy parameters

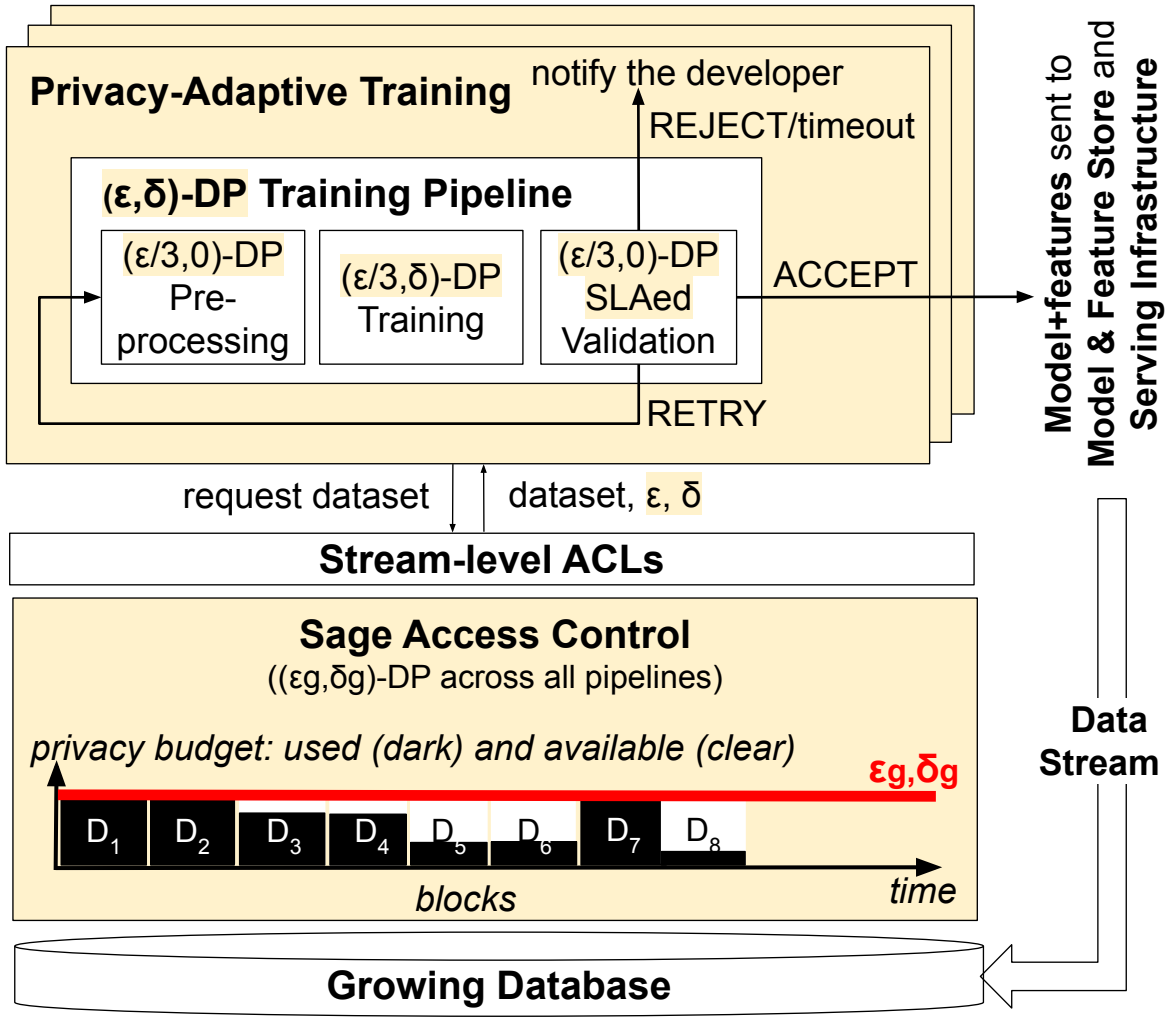


Figure 2.2: Sage DP ML Platform. Highlights changes from non-DP version.

given by Sage at runtime (box (ϵ, δ)-*DP Training Pipeline*, §2.3). The developer is responsible for making this switch to DP, and while research is needed to ease DP programming, this paper leaves that challenge aside.

Second, Sage introduces an additional layer of access control beyond traditional stream-level ACLs (box *Sage Access Control*, §2.3). The new layer splits the data stream into *blocks* and accounts for the privacy loss of releasing a model+features bundle at the level of the specific blocks that were used to train that bundle. In theory, blocks can be defined by any insensitive attribute, with two attributes particularly relevant here: time (e.g., one day's worth of data goes into one block) and user ID (e.g., a user's data goes into the same block). Defin-

ing blocks by time provides event-level privacy; defining them by user ID accommodates user-level privacy. Because of our focus on the former, the remainder of this section assumes that blocks are defined by time; §2.3 discusses sharding by user ID and other attributes.

When the privacy loss for a block reaches the (ϵ_g, δ_g) ceiling, the block is *retired* (blocks D_1, D_2, D_7 are retired in Figure 2.2). However, new blocks arrive with a clean budget and can be used to train future models: *as long as the database grows fast enough in new blocks*, the system will never run out of privacy budget for the stream. Perhaps surprisingly, this privacy loss accounting method, which we call *block composition*, is the first practical approach to avoid running out of privacy budget while enabling effective training of ML models on growing databases. §2.3 gives the intuition of block composition while §2.3 formalizes it and proves it (ϵ_g, δ_g)-DP.

Third, Sage provides developers with control over the quality of models produced by the DP Training Pipelines. Such pipelines can produce less accurate models that fail to meet their quality targets more often than without DP. They can also push in production low-quality models whose validations succeed by mere chance. Both situations lead to operational headaches: the former gives more frequent notifications of failed training, the latter gives dissatisfied users. The issue is often referred to as the *privacy-utility tradeoff* of running under a DP regime. Sage addresses this challenge by wrapping the (ϵ, δ)-DP Training Pipeline into an iterative process that reduces the effects of DP randomness on the quality of the models and the semantic of their validation by invoking training pipelines repeatedly on increasing amounts of data and/or privacy budgets (box *Privacy-Adaptive Training*, §2.3).

Example (ϵ, δ)-DP Training Pipeline

Sage expects each pipeline submitted by the ML developer to satisfy a parameterized (ϵ, δ)-DP. Acknowledging that DP programming abstractions warrant further research, Listing 2.1 illustrates the changes a developer would have to make at present to convert a non-DP training

```
def preprocessing_fn(inputs, epsilon):
  dist_01 = tft.scale_to_0_1(inputs["distance"],0,100)
  speed_01 = tft.scale_to_0_1(inputs["speed"],0,100)
  hour_of_day_speed = group_by_mean
  sage.dp_group_by_mean(
    inputs["hour_of_day"], speed_01, 24, epsilon, 1.0)
  return {"dist_scaled": dist_01,
    "hour_of_day": inputs["hour_of_day"],
    "hour_of_day_speed": hour_of_day_speed,
    "duration": inputs["duration"]}

def trainer_fn(hparams, schema, epsilon, delta): [...]
  feature_columns = [numeric_column("dist_scaled"),
    numeric_column("hour_of_day_speed"),
    categorical_column("hour_of_day", num_buckets=24)]
  estimator = \
    tf.estimator.DNNRegressorsage.DPDNNRegressor(
      config=run_config,
      feature_columns=feature_columns,
      dnn_hidden_units=hparams.hidden_units,
      privacy_budget=(epsilon, delta))
  return tfx.executors.TrainingSpec(estimator,...)

def validator_fn(epsilon):
  model_validator = \
    tfx.components.ModelValidatorsage.DPModelValidator(
      examples=examples_gen.outputs.output,
      model=trainer.outputs.output,
      metric_fn = _MSE_FN, target = _MSE_TARGET,
      epsilon=epsilon, confidence=0.95, B=1)
  return model_validator

def dp_group_by_mean(key_tensor, value_tensor, nkeys,
  epsilon, value_range):
  key_tensor = tf.dtypes.cast(key_tensor, tf.int64)
  ones = tf.fill(tf.shape(key_tensor), 1.0)
  dp_counts = group_by_sum(key_tensor, ones, nkeys)\
    + laplace(0.0, 2/epsilon, nkeys)
  dp_sums = group_by_sum(
    key_tensor,value_tensor,nkeys)\
    + laplace(0.0, value_range * 2/epsilon, nkeys)
  return tf.gather(dp_sums/dp_counts, key_tensor)
```

Listing 2.1: Example Training Pipeline. Shows non-DP TFX (stricken through) and DP Sage (highlighted) versions. TFX API is simplified for exposition.

pipeline written for TFX to a DP training pipeline suitable for Sage. Removed/replaced code is stricken through and the added code is highlighted. The pipeline processes New York City Yellow Cab data[8] and trains a model to predict the duration of a ride.

To integrate with TFX (non-DP version), the developer implements three TFX callbacks. (1) `preprocessing_fn` uses the dataset to compute aggregate features and make user-specified feature transformations. The model has three features: the distance of the ride; the hour of the day; and an aggregate feature representing the average speed of cab rides each

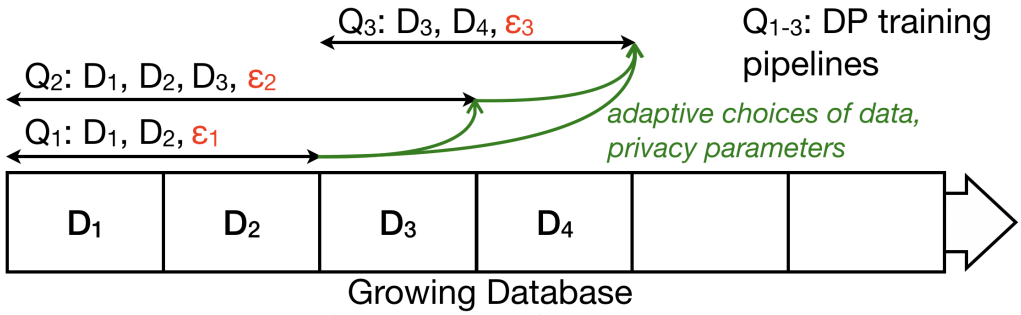


Figure 2.3: Characteristics of Data Interaction in ML.

hour of the day. (2) `trainer_fn` specifies the model that is to be trained: it configures the columns to be modeled, defines hyperparameters, and specifies the dataset. The model trains with a neural network regressor. (3) `validator_fn` validates the model by comparing test set MSE to a constant.

To integrate with Sage (DP version), the developer: (a) switches library calls to DP versions of the functions (which ideally would be available in the ML platform) and (b) splits the (ϵ, δ) parameters, which are assigned by Sage at runtime, across the DP function calls. (1) `preprocessing_fn` replaces one call with a DP version that is implemented in Sage: the mean speed per day uses Sage's `dp_group_by_mean`. This function (lines 32-40) computes the number of times each key appears and the sum of the values associated with each key. It makes both DP by adding draws from appropriately-scaled Laplace distributions to each count. Each data point has exactly one key value so the privacy budget usage composes in parallel across keys[111]. The privacy budget is split across the sum and count queries. We envision common functions like this being available in the DP ML platform. (2) `trainer_fn` switches the call to the non-private regressor with the DP implementation, which in Sage is a simple wrapper around TensorFlow's DP SGD-based optimizer. (3) `validator_fn` invokes Sage's DP model validator (§2.3).

Sage Access Control

Sage uses the composition property of DP to rigorously account for (and bound) the cumulative leakage of data from sensitive user streams across multiple releases of models+features

learned from these streams. Specifically, for each sensitive stream, Sage maintains a pre-specified event-level (ϵ_g, δ_g)-DP guarantee across all uses of the stream. Unfortunately, traditional DP composition theory considers either static databases, which leads to wasteful privacy accounting; or purely online streaming, which is inefficient for many ML workloads, including deep neural network training. We thus developed our own composition theory, called *block composition*, which leverages characteristics of ML workloads running on growing databases to permit both efficient privacy accounting and efficient learning. §2.3 formalizes the new theory. This section describes the limitations of existing DP composition for ML and gives the intuition for block composition and how Sage uses it as a new form of access control in ML platforms.

Data Interaction in ML on Growing Databases. Figure 2.3 shows an example of a typical workload as seen by an ML platform. Each training pipeline, or *query* in DP parlance, is denoted Q_i. We note two characteristics. First, a typical ML workload consists of multiple training pipelines, training over time on a continuously growing database, and on data subsets of various sizes. For instance, Q_2 may train a large deep neural network requiring massive amounts of data to reach good performances, while Q_3 may train a linear model with smaller data requirements, or even a simple statistic like the mean of a feature over the past day. All pipelines are typically updated or retrained periodically on new data, with old data eventually being deemed irrelevant and ignored.

Second, the data given to a training pipeline – and for a DP model its DP parameters – are typically chosen *adaptively*. For example, the model trained in Q_1 on data from $D_{1,2}$ with budget ϵ_1 may give unsatisfactory performance. After a new block D_3 is collected, a developer may decide to retrain the same model in query Q_2 on data from $D_{1,2,3}$, and with a higher DP budget ϵ_2. Adaptivity can also happen indirectly through the data. Suppose Q_2 successfully trained a recommendation model. Then, future data collected from the users (e.g. in D_4) may depend on the recommendations. Any subsequent query, such as Q_3, is

potentially influenced by Q_2's output.

These characteristics imply three requirements for a composition theory suitable for ML. It must support:

- **R1** Queries on overlapping data subsets of diverse sizes.
- **R2** Adaptivity in the choice of: queries, DP parameters, and data subsets the queries process.
- **R3** Endless execution on a growing database.

Limitations of Existing Composition Theory for ML. No previous DP composition theory supports all three requirements. DP has mostly been studied for static databases, where (adaptively chosen) queries are assumed to compute over *the entire database*. Consequently, composition accounting is typically made at *query level*: each query consumes part of the total available privacy budget for the database. Query-level accounting has carried over even in extensions to DP theory that handle streaming databases[55] and partitioned queries[111]. There are multiple ways to apply query-level accounting to ML, but each trades off at least one of our requirements.

First, one can query overlapping data subsets (**R1**) adaptively across queries, the data used, and the DP parameters[135] (**R2**) by accounting for composition at the query level *against the entire stream*. Queries on Figure 2.3 would thus result in a total privacy loss of $\epsilon_1 + \epsilon_2 + \epsilon_3$ over the whole stream. This approach wastes privacy budget and leads to the problem of "running out of budget". Once $\epsilon_g = \epsilon_1 + \epsilon_2 + \epsilon_3$, enforcing a global leakage bound of ϵ_g means that *one must stop using the stream* after query Q_3. This is true even though (1) not all queries run on all the data and (2) there will be new data coming into the system in the future (e.g., D_5). This violates requirement (**R3**) of endless execution on streams.

Second, one can restructure the queries to enable finer granularity with query-level accounting. The data stream is partitioned in blocks, as in Figure 2.3. Each query is split into multiple ones, each running with DP on an individual block. The DP results are then ag-

gregated, for instance by averaging model updates as in federated learning[109]. Since each block is a separate dataset, traditional composition can account for privacy loss at the block level. This approach supports adaptivity (**R2**) and execution of the system on streams (**R3**) as new data blocks incur no privacy loss from past queries. However, it violates requirement (**R1**), resulting in unnecessarily noisy learning[47;48]. Consider computing a feature average. DP requires adding noise once, after summing all values on the combined blocks. But with independent queries over each block, we must add the same amount of noise to the sum over each block, yielding a more noisy total. Additionally, several DP training algorithms[10;109] fundamentally rely on sampling small training batches from large datasets to amplify privacy, which cannot be done without combining blocks.

Third, one can consume the data stream online using streaming DP. A new data point is allocated to one of the waiting queries, which consumes its entire privacy budget. Because each point is used by one query and discarded, DP holds over the entire stream. New data can be adaptively assigned to any query (**R2**) and arrives with a fresh budget (**R3**). However, queries cannot use past data or overlapping subsets, violating **R1** and rendering the approach impractical for large models.

Block Composition. Our new composition theory meets all three requirements. It splits the data stream into disjoint *blocks* (e.g., one day's worth of data for event-level privacy), forming a growing database on which queries can run on overlapping and adaptively chosen sets of blocks (**R1**, **R2**). This lets pipelines combine blocks with available privacy budgets to assemble large datasets. Despite running overlapping queries, we can still account for the privacy loss of each individual blocks, where each query impacts the blocks it actually uses, *not the entire data stream*. Unused blocks, including future ones, incur no privacy loss. In Figure 2.3, the first three blocks each incur a privacy loss of $\epsilon_1 + \epsilon_2$ while the last block has $\epsilon_2 + \epsilon_3$. The privacy loss of these three queries over the entire data stream will only be the maximum of these two values. Moreover, when the database grows (e.g. block D_5 arrives),

the new blocks' privacy loss is zero. The system can thus run endlessly by training new models on new data (**R3**).

Sage Access Control. With block composition, Sage controls data leakage from a stream by enforcing DP on its blocks. The company configures a desirable (ϵ_g, δ_g) global policy for each sensitive stream. The Sage Access Control component tracks the available privacy budget for each data block. It allows access to a block until it runs out of budget, after which access to the block will never be granted again. When the Sage Iterator (described in §2.3) for a pipeline requests data, Sage Access Control only offers blocks with available privacy budget. The Iterator then determines the (ϵ, δ) privacy parameters it will use for its iteration and informs Sage Access Control, which deducts (ϵ, δ) from the available privacy budgets of those blocks. Finally, the Iterator invokes the developer-supplied DP Training Pipeline, trusting it to enforce the chosen (ϵ, δ) privacy parameters. §2.3 proves that this access control policy enforces (ϵ_g, δ_g)-DP for the stream.

The preceding operation is a DP-informed retention policy, but one can use block composition to define other access control policies. Suppose the company is willing to assume that its developers (or user devices and prediction servers in distinct geographies) will not collude to violate its customers' privacy. Then the company could enforce a separate (ϵ_g, δ_g) guarantee for each context (developer or geography) by maintaining separate lists of per-block available budgets.

Privacy-Adaptive Training

Sage's design adds reliability to the DP model training and validation processes, which are rendered imprecise by the DP randomness. We describe two novel techniques: (1) *SLAed validation*, which accounts for the effect of randomness in the validation process to ensure a high-probability guarantee of correctness (akin to a quality service level agreement, or SLA); and (2) *privacy-adaptive training*, which launches the (ϵ, δ)-DP Training Pipeline adaptively

```
class DPLossValidator(sage.DPModelValidator):
  def validate(loss_fn, target, epsilon, conf, B):
    if _ACCEPT_test(..., epsilon, (1-conf)/2, B):
      return ACCEPT
    if _REJECT_test(..., epsilon, (1-conf)/2, B):
      return REJECT
    return RETRY

  def _ACCEPT_test(test_labels, dp_test_predictions,
        loss_fn, target, epsilon, eta, B):
    n_test = dp_test_predictions.size()
    n_test_dp = n_test + laplace(2/epsilon)
    n_test_dp_min = n_test_dp -\
      2*log(3/(2*eta))/epsilon
    dp_test_loss = clip_by_value(loss_fn(test_labels,
      dp_test_predictions), 0, B)+laplace(2*B/epsilon)
    corrected_dp_test_loss = dp_test_loss +
      2*B*log(3/(2*eta))/epsilon
    return bernstein_upper_bound(
      corrected_dp_test_loss / n_test_dp_min,
      n_test_dp_min, eta/3, B) <= target

  def bernstein_upper_bound(loss, n, eta, B):
    return loss+sqrt(2*B*loss*log(1/eta)/n)+\
      4*log(1/eta)/n
```

Listing 2.2: Implementation of `sage.DPLossValidator`.

on increasing amounts of data from the stream, and/or with increased privacy parameters, until the validation succeeds. Privacy-adaptive training thus leverages the adaptivity properties of block composition to address DP's privacy-utility tradeoff.

SLAed DP Validation. Figure 2.2 shows the three possible outcomes of SLAed validation: ACCEPT, REJECT/timeout, and RETRY. If SLAed validation returns ACCEPT, then with high probability (e.g. 95%) the model reached its configured quality targets for prediction on new data from the same distribution. Under certain assumptions, it is also possible to give statistical guarantees of correct negative assessment, in which case SLAed validation returns REJECT. We refer the reader to our extended technical report[98] for this discussion. Sage also supports timing out a training procedure if it has run for too long. Finally, if SLAed validation returns RETRY, it signals that more data is needed for an assessment.

We have implemented SLAed validators for three classes of metrics: loss metrics (e.g. MSE, log loss), accuracy, and *absolute errors* of sum-based statistics such as mean and variance. The technical report[98] details the implementations and proves their statistical and DP guarantees. Here, we give the intuition and an example based on loss metrics. All validators

follow the same logic. First, we compute a DP version of the test quantity (e.g. MSE) on a testing set. Second, we compute the *worst-case impact of DP noise* on that quantity for a given confidence probability; we call this a *correction for DP impact*. For example, if we add Laplace noise with parameter $\frac{1}{\epsilon}$ to the sum of squared errors on n data points, assuming that the loss is in $[0, 1]$ we know that with probability $(1 - \eta)$ the sum is deflated by less than $-\frac{1}{\epsilon} \ln(\frac{1}{2\eta})$, because a draw from this Laplace distribution has just an η probability to be more negative than this value. Third, we use known statistical concentration inequalities, also made DP and corrected for worst case noise impact, to upper bound with high probability the loss on the entire distribution.

Example: Loss SLAed Validator. A loss function is a measure of erroneous predictions on a dataset (so lower is better). Examples include: mean squared error for regression, log loss for classification, and minus log likelihood for Bayesian generative models. Listing 2.2 shows our loss validator. The validation function consists of two tests: ACCEPT (described here) and REJECT (described in the technical report[98]).

Denote: the DP-trained model f^{dp}; the loss function range $[0, B]$; the target loss τ_{loss}. Lines 11-13 compute a DP estimate of the number of samples in the test set, corrected for the impact of DP noise to be a lower bound on the true value with probability $(1 - \frac{\eta}{3})$. Lines 14-17 compute a DP estimate of the loss sum, corrected for DP impact to be an upper bound on the true value with probability $(1 - \frac{\eta}{3})$. Lines 18-20 ACCEPT the model if the upper bound is at most τ_{loss}. The bounds are based on a standard concentration inequality (specifically, Bernstein's inequality), which holds under very general conditions[151]. We show in[98] that the Loss ACCEPT Test satisfies $(\epsilon, 0)$-DP and enjoys the following guarantee:

Proposition 1 (Loss ACCEPT Test)**.** With probability at least $(1-\eta)$, the ACCEPT test returns true only if the expected loss of f^{dp} is at most τ_{loss}.

Privacy-Adaptive Training. Sage attempts to improve the quality of the model and its validation by supplying them with more data or privacy budgets so the SLAed validator can

either ACCEPT or REJECT the model. Several ways exist to improve a DP model's quality. First, we can increase the dataset's size: at least in theory, it has been proven that one can compensate for the loss in accuracy due to *any* (ϵ, δ)-DP guarantee by increasing the training set size[91]. Second, we can increase the privacy budget (ϵ, δ) to decrease the noise added to the computation: this must be done within the available budgets of the blocks involved in the training and *not too aggressively*, because wasting privacy budget on one pipeline can prevent other pipelines from using those blocks.

Privacy-adaptive training searches for a configuration that can be either ACCEPTed or REJECTed by the SLAed validator. We have investigated several strategies for this search. Those that conserve privacy budget have proven the most efficient. Every time a new block is created, its budget is divided evenly across the ML pipelines currently waiting in the system. Allocated DP budget is reserved for the pipeline that received it, but privacy-adaptive training will not use all of it right away. It will try to ACCEPT using as little of the budget as possible. When a pipeline is ACCEPTed, its remaining budget is reallocated evenly across the models still waiting in Sage.

To conserve privacy budget, each pipeline will first train and test using a small configurable budget (ϵ_0, δ_0), and a minimum window size for the model's training. On RETRY from the validator, the pipeline will be retrained, making sure to double either the privacy budget if enough allocation is available to the Training Pipeline, or the number of samples available to the Training Pipeline by accepting new data from the stream. This doubling of resources ensures that when a model is ACCEPTed, the sum of budgets used by all failed iterations is at most equal to the budget used by the final, accepted iteration. This final budget also overshoots the best possible budget by at most two, since the model with half this final budget had a RETRY. Overall, the resources used by this DP budget search are thus at most four times the budget of the final model. Evaluation §2.4 shows that this conservative strategy improves performance when multiple Training Pipelines contend for the same blocks.

Block Composition Theory

This section provides the theoretical backing for block composition, which we invent for Sage but which we believe has broader applications (§2.3). To analyze composition, one formalizes permissible interactions with the sensitive data in a *protocol* that facilitates the proof of the DP guarantee. This interaction protocol makes explicit the worst-case decisions that can be made by modeling them through an adversary. In the standard protocol (detailed shortly), an adversary $\mathcal{A}$ picks the neighboring data sets and supplies the DP queries that will compute over one of these data sets; the choice between the two data sets is exogenous to the interaction. To prove that the interaction satisfies DP, one must show that given the results of the protocol, it is impossible to determine with high confidence which of the two neighboring data sets was used.

Figure 2.7 describes three different interaction protocols of increasing sophistication. Alg. (2.4) is the basic DP composition protocol. Alg. (2.5) is a block-level protocol we propose for static databases. Alg. (2.6) is the protocol adopted in Sage; it extends Alg. (2.5) by allowing a streaming database and adaptive choices of blocks and privacy parameters. Highlighted are changes made to the preceding protocol.

Traditional Query-Level Accounting

QueryCompose (Alg. (2.4)) is the interaction protocol assumed in most analyses of composition of several DP interactions with a database. There are three important characteristics. First, the number of queries r and the DP parameters $(\epsilon_i, \delta_i)_{i=1}^r$ are fixed in advance. However the DP queries $\mathcal{Q}_i$ can be chosen adaptively. Second, the adversary adaptively chooses neighboring datasets $\mathcal{D}^{i,0}$ and $\mathcal{D}^{i,1}$ for each query. This flexibility lets the protocol readily support adaptively evolving data (such as with data streams) where future data collected may be impacted by the adversary's change to past data. Third, the adversary receives the results V^b of running the DP queries $\mathcal{Q}_i$ on $\mathcal{D}^{i,b}$; here, $b \in \{0, 1\}$ is the exogenous choice of which

database to use and is unknown to $\mathcal{A}$. DP is guaranteed if $\mathcal{A}$ cannot confidently learn b given V^b.

A common tool to analyze DP protocols is *privacy loss*:

Definition 2 (Privacy Loss)**.** Fix any outcome $v = (v_1, \ldots, v_r)$ and denote $v_{<i} = (v_1, \ldots, v_{i-1})$. The *privacy loss* of an algorithm Compose($\mathcal{A}$, b, r, $\cdot$) is:

$$\text{Loss}(v) = \ln\Big(\frac{P(V^0 = v)}{P(V^1 = v)}\Big) = \ln\Big(\prod_{i=1}^{r} \frac{P(V_i^0 = v_i | v_{<i})}{P(V_i^1 = v_i | v_{<i})}\Big)$$

Bounding the privacy loss for any adversary $\mathcal{A}$ with high probability implies DP[92]. Suppose that for any $\mathcal{A}$, with probability $\geq (1 - \delta)$ over draws from $v \sim V^0$, we have: $|\text{Loss}(v)| \leq \epsilon$. Then Compose($\mathcal{A}$, b, r, $\cdot$) is (ϵ, δ)-DP. This way, privacy loss and DP are defined in terms of distinguishing between two hypotheses indexed by $b \in \{0, 1\}$.

Previous composition theorems (e.g. basic composition[53], strong composition[58], and variations thereof[90]) analyze Alg. (2.4) to derive various arithmetics for computing the overall DP semantic of interactions adhering to that protocol. In particular, the basic composition theorem[53] proves that QueryCompose($\mathcal{A}$, b, r, $(\epsilon_i, \delta_i)_{i=1}^r$) is $(\sum_{i=1}^r \epsilon_i, \sum_{i=1}^r \delta_i)$-DP. These theorems form the basis of most ML DP work. However, because composition is accounted for at the query level, imposing a fixed global privacy budget means that one will "run out" of it and stop training models even on new data.

Block Composition for Static Datasets

Block composition improves privacy accounting for workloads where interaction consists of queries that run on *overlapping data subsets of diverse sizes*. This is one of the characteristics we posit for ML workloads (requirement **R1** in §2.3). Alg. (2.5), *BlockCompose*, formalizes this type of interaction for a *static dataset setting* as a springboard to formalizing the full ML interaction. We make two changes to QueryCompose. First (line 1), the neighboring datasets are defined once and for all before interacting. This way, training pipelines accessing non-

overlapping parts of the dataset cannot all be impacted by one entry's change. Second (line 4), the data is split in blocks, and each DP query runs on a subset of the blocks.

We prove that the privacy loss over the entire dataset is the same as the maximum privacy loss on each block, accounting only for queries using this block:

Theorem 1 (Reduction to Block-level Composition)**.** The privacy loss of BlockCompose($\mathcal{A}$, b, r, $(\epsilon_i, \delta_i)_{i=1}^r$, $(\text{blocks}_i)_{i=1}^r$) is upper-bounded by the maximum privacy loss for any block:

$$|\operatorname{Loss}(v)| \leq \max_k \Big|\ln\Big(\prod_{\substack{i=1\\ k\in\text{blocks}_i}}^{r} \frac{P(V_i^0 = v_i|v_{<i})}{P(V_i^1 = v_i|v_{<i})}\Big)\Big|.$$

Proof. Let $\mathcal{D}^0$ and $\mathcal{D}^1$ be the neighboring datasets picked by adversary $\mathcal{A}$, and let k be the block index s.t. $\mathcal{D}_l^0 = \mathcal{D}_l^1$ for all $l \neq k$, and $|\mathcal{D}_k^0 \oplus \mathcal{D}_k^1| \leq 1$. For any result v of Alg. (2.5):

$$\begin{aligned}
|\operatorname{Loss}(v)| &= \Big|\ln\Big(\prod_{i=1}^{r} \frac{P(V_i^0 = v_i|v_{<i})}{P(V_i^1 = v_i|v_{<i})}\Big)\Big| \\
&= \Big|\ln\Big(\prod_{\substack{i=1\\ k\in\text{blocks}_i}}^{r} \frac{P(V_i^0 = v_i|v_{<i})}{P(V_i^1 = v_i|v_{<i})}\Big) + \cancel{\ln\Big(\prod_{\substack{i=1\\ k\notin\text{blocks}_i}}^{r} \frac{P(V_i^0 = v_i|v_{<i})}{P(V_i^1 = v_i|v_{<i})}\Big)}\Big| \\
&\leq \max_k \Big|\ln\Big(\prod_{\substack{i=1\\ k\in\text{blocks}_i}}^{r} \frac{P(V_i^0 = v_i|v_{<i})}{P(V_i^1 = v_i|v_{<i})}\Big)\Big|
\end{aligned}$$

The slashed term is zero because if $k \notin \text{blocks}_i$, then

$\bigcup_{j\in\text{blocks}_i} \mathcal{D}_j^0 = \bigcup_{j\in\text{blocks}_i} \mathcal{D}_j^1$, hence $\frac{P(V_i^0=v_i|v_{<i})}{P(V_i^1=v_i|v_{<i})} = 1$. □

Hence, unused data blocks allow training of other (adaptively chosen) ML models, and exhausting the DP budget of a block means we retire that block of data, and not the entire data set. This result, which can be extended to strong composition (see tech. report[98]), can be used to do tighter accounting than query-level accounting when the workload consists of queries on overlapping sets of data blocks (requirement **R1**). However, it does not support adaptivity in block choice or a streaming setting, violating **R2** and **R3**.

Sage Block Composition

Alg. (2.6), *AdaptiveStreamBlockCompose*, addresses the limitations with two changes. First, supporting streams requires that datasets not be fixed before interacting, because future data depends on prior models trained and pushed into production. The highlighted portions of lines 1-10 in Alg. (2.6) formalize the dynamic nature of data collection by having new data blocks explicitly depend on previously trained models, which are chosen by the adversary, in addition to other mechanisms of the world $\mathcal{W}$ that are not impacted by the adversary. Fortunately, Theorem 1 still applies, because model training can only use blocks that existed at the time of training, which in turn only depend on prior blocks through DP trained models. Therefore, new data blocks can train new ML models, enabling endless operation on streams (**R3**).

Second, interestingly, supporting adaptive choices in the data blocks implies supporting adaptive choices in the queries' DP budgets. For a given block, one can express query i's choice to use block j as using a privacy budget of either (ϵ_i, δ_i) or $(0, 0)$. Lines 7-8 in Alg. (2.6) formalize the adaptive choice of both privacy budgets and blocks (requirement **R2**). It does so by leveraging recent work on DP composition under adaptive DP budgets[135]. At each round, $\mathcal{A}$ requests access to a group of blocks blocks_i, on which to run an (ϵ_i, δ_i)-DP query. Sage's Access Control permits the query only if the privacy loss of each block in blocks_i will remain below (ϵ_g, δ_g). Applying our Theorem 1 and[135]'s Theorem 3.3, we prove the following result (proof in[98]):

Theorem 2 (Composition for Sage Block Composition)**.** AdaptiveStreamBlockCompose($\mathcal{A}$, b,r,ϵ_g,δ_g,$\mathcal{W}$) is (ϵ_g, δ_g)-DP if for all k, AccessControl$^k_{\epsilon_g,\delta_g}$ enforces:

$$\Big(\sum_{\substack{i=1 \\ k \in \text{blocks}_i}}^{r} \epsilon_i(v_{<i}) \Big) \leq \epsilon_g \text{ and } \Big(\sum_{\substack{i=1 \\ k \in \text{blocks}_i}}^{r} \delta_i(v_{<i}) \Big) \leq \delta_g.$$

The implication of the preceding theorem is that under the access control scheme described in §2.3, Sage achieves *event-level* (ϵ_g, δ_g)*-DP over the sensitive data stream.*

Blocks Defined by User ID and Other Attributes

Block composition theory can be extended to accommodate user-level privacy and other use cases. The theory shows that one can split a static dataset (Theorem 1) or a data stream (Theorem 2) into disjoint blocks, and run DP queries adaptively on overlapping subsets of the blocks while accounting for privacy at the block level. The theory focused on *time* splits, but the same theorems can be written for splits based on *any attribute whose possible values can be made public*, such as geography, demographics, or user IDs. Consider a workload on a static dataset in which queries combine data from diverse and overlapping subsets of countries, e.g., they compute average salary in each country separately, but also at the level of continents and GDP-based country groups. For such a workload, block composition gives tighter privacy accounting across these queries than traditional composition, though the system will still run out of privacy budget eventually because no new blocks appear in the static database.

As another example, splitting a stream by user ID enables querying or ignoring all observations from a given user, adding support for user-level privacy. Splitting data over user ID requires extra care. If existing user IDs are not knows, each query might select user IDs that do not exist yet, spending their DP budget without adding data. However, making user IDs public can leak information. One approach is to use incrementing user IDs (with this fact public), and periodically run a DP query computing the maximum user ID is use. This would ensure DP, while giving an estimate of the range of user IDs that can be queried. In such a setting, block composition enables fine-grain DP accounting over queries on any subset of the users. While our block theory supports this use case, it suffers from a major practical challenge. New blocks are now created only when new users join the system, so new users must be added at a high rate relative to the model release rate to avoid running out of budget. This is unlikely to happen for mature companies, but may be possible for emerging startups or hospitals, where the stream of incoming users/patients may be high enough to sustain

Taxi Regression Task		
Pipelines:	**Configuration:**	
Linear Regression (**LR**)	DP Alg.	AdaSSP from[172], (ϵ, δ)-DP
	Config.	Regularization param ρ : 0.1
	Budgets	$(\epsilon, \delta) \in \{(1.0, 10^{-6}), (0.05, 10^{-6})\}$
	Targets	MSE $\in [2.4 \times 10^{-3}, 7 \times 10^{-3}]$
Neural Network (**NN**)	DP Alg.	DP SGD from[10], (ϵ, δ)-DP
	Config.	ReLU, 2 hidden layers (5000/100 nodes) Learning rate: 0.01, Epochs: 3 Batch: 1024, Momentum: 0.9
	Budgets	$(\epsilon, \delta) \in \{(1.0, 10^{-6}), (0.5, 10^{-6})\}$
	Targets	MSE $\in [2 \times 10^{-3}, 7 \times 10^{-3}]$
Avg.Speed x3*	Targets	Absolute error $\in \{1, 5, 7.5, 10, 15\}$ km/h
Criteo Classification Task		
Pipelines:	**Configuration:**	
Logistic Regression (**LG**)	DP Alg.	DP SGD from[107], (ϵ, δ)-DP
	Config.	Learning rate: 0.1, Epochs: 3 Batch: 512
	Budgets	$(\epsilon, \delta) \in \{(1.0, 10^{-6}), (0.25, 10^{-6})\}$
	Targets	Accuracy $\in [0.74, 0.78]$
Neural Network (**NN**)	DP Alg.	DP SGD from[107], (ϵ, δ)-DP
	Config.	ReLU, 2 hidden layers (1024/32 nodes) Learning rate: 0.01, Epochs: 5 Batch: 1024
	Budgets	$(\epsilon, \delta) \in \{(1.0, 10^{-6}), (0.25, 10^{-6})\}$
	Targets	Accuracy $\in [0.74, 0.78]$
Counts x26**	Targets	Absolute error $\in \{0.01, 0.05, 0.10\}$

Table 2.1: Experimental Training Pipelines. *Three time granularities: hour of day, day of week, week of month. **Histogram of each categorical feature.

modest workloads.

2.4 Evaluation

We ask four questions: (**Q1**) Does DP impact Training Pipeline reliability? (**Q2**) Does privacy-adaptive training increase DP Training Pipeline reliability? (**Q3**) Does block composition help over traditional composition? (**Q4**) How do ML workloads perform under Sage's (ϵ_g, δ_g)-DP regime?

Methodology. We consider two datasets: 37M-samples from three months of NYC taxi rides[8] and 45M ad impressions from Criteo[3]. On the Taxi dataset we define a regression task to predict the duration of each ride using 61 binary features derived from 10 contextual

features. We implement pipelines for a *linear regression (LR)*, a *neural network (NN)*, and three statistics (average speeds at three time granularities). On the Criteo dataset we formulate a binary classification task predicting ad clicks from 13 numeric and 26 categorical features. We implement a *logistic regression (LG)*, a *neural network (NN)*, and histogram pipelines. Table 2.1 shows details.

Training: We make each pipeline DP using known algorithms, shown in Table 2.1. *Validation:* We use the loss, accuracy, and absolute error SLAed validators on the regression, classification, and statistics respectively. *Experiments:* Each model is assigned a quality target from a range of possible values, chosen between the best achievable model, and the performance of a naïve model (predicting the label mean on Taxi, with MSE 0.0069, and the most common label on Criteo, with accuracy 74.3%). Most evaluation uses privacy-adaptive training, so privacy budgets are chosen by Sage, with an upper-bound of $\epsilon = 1$. While no consensus exists on what a reasonable DP budget is, this value is in line with practical prior work[10;109]. Where DP budgets must be fixed, we use values indicated in Table 2.1 which correspond to a large budget ($\epsilon = 1$), and a small budget that varies across tasks and models. Other defaults: 90%::10% train::test ratio; $\eta = 0.05$; $\delta = 10^{-6}$. *Comparisons:* We compare Sage's performance to existing DP composition approaches described in §2.3. We ignore the first alternative, which violates the endless execution requirement **R3** and cannot support ML workloads. We compare with the second and third alternatives, which we call *query composition* and *streaming composition*, respectively.

Unreliability of DP Training Pipelines in TFX (Q1)

We first evaluate DP's impact on model training. Figure 2.8 shows the loss or accuracy of each model when trained on increasing amounts data and evaluated on 100K held-out samples from their respective datasets. Three versions are shown for each model: the non-DP version (NP), a large DP budget version ($\epsilon = 1$), and a small DP budget configuration

Dataset	η	No SLA	NP SLA	UC DP SLA	Sage SLA
Taxi	0.01	0.379	0.0019	0.0172	0.0027
	0.05	0.379	0.0034	0.0224	0.0051
Criteo	0.01	0.2515	0.0052	0.0544	0.0018
	0.05	0.2515	0.0065	0.0556	0.0023

Table 2.2: Target Violation Rate of ACCEPTed Models. Violations are across all models separately trained with privacy-adaptive training.

with ϵ values that vary across the model and task. For both tasks, the NN requires the most data but outperforms the linear model in the private and non-private settings. The DP LRs catch up to the non-DP version with the full dataset, but the other models trained with SGD require more data. Thus, model quality is impacted by DP but the impact diminishes with more training data. This motivates privacy-adaptive training.

To evaluate DP's impact on validation, we train and validate our models for both tasks, with and without DP. We use TFX's vanilla validators, which compare the model's performance on a test set to the quality metric (MSE for taxi, accuracy for Criteo). We then re-evaluate the models' quality metrics on a separate, 100K-sample held-out set and measure the fraction of models accepted by TFX that violate their targets on the re-evaluation set. With non-DP pipelines (non-DP training and validation), the false acceptance rate is 5.1% and 8.2% for the Taxi and Criteo tasks respectively. With DP pipelines (DP training, DP validation), false acceptance rates hike to 37.9% and 25.2%, motivating SLAed validation.

Reliability of DP Training Pipelines in Sage (Q2)

Sage's privacy-adaptive training and SLAed validation are designed to add reliability to DP model training and validation. However, they may come at a cost of increased data requirements over a non-DP test. We evaluate reliability and sample complexity for the SLAed validation ACCEPT test.

Table 2.2 shows the fraction of ACCEPTed models that violate their quality targets when re-evaluated on the 100K-sample held-out set. For two confidences η, we show: (1) *No SLA*, the vanilla TFX validation with no statistical rigor, but where a model's quality is computed

with DP. (2) *NP SLA*, a non-DP but statistically rigorous validation. This is the best we can achieve with statistical confidence. (3) *UC DP SLA*, a DP SLAed validation without the correction for DP impact. (4) *Sage SLA*, our DP SLAed validator, with correction. We make three observations. First, the NP SLA violation rates are much lower than the configured η values because we use conservative statistical tests. Second, Sage's DP-corrected validation accepts models with violation rates close to the NP SLA. Slightly higher for the loss SLA and slightly lower for the accuracy SLA, but *well below the configured error rates*. Third, removing the correction increases the violation rate by 5x for the loss SLA and 20x for the accuracy SLA, violating the confidence thresholds in both cases, at least for low η. These results confirm that Sage's SLAed validation is reliable, and that correction for DP is critical to this reliability.

The increased reliability of SLAed validation comes at a cost: SLAed validation requires more data compared to a non-DP test. This new data is supplemented by Sage's privacy-adaptive training. Figure 2.9(a) and 2.9(b) show the amount of train+test data required to ACCEPT a model under various loss targets for the Taxi regression task. Figure 2.9(c) and 2.9(d) show the same for accuracy targets for the Criteo classification task. We make three observations. First, unsurprisingly, non-rigorous validation (No SLA) requires the least data but has a high failure rate because it erroneously accepts models on small sample sizes. Second, the best model accepted by Sage's SLA validation are close to the best model accepted by No SLA. We observe the largest difference in Taxi LR where No SLA accepts MSE targets of 0.0025 while the Sage SLA accepts as low as 0.0027. The best achievable model is slightly impacted by DP, although more data is required. Third, adding a statistical guarantee but no privacy to the validation (NP SLA) already substantially increases sample complexity. Adding DP to the statistical guarantee and applying the DP correction incurs limited additional overhead. The distinction between Sage and NP SLA is barely visible for all but the Taxi LR. For Taxi LR, adding DP accounts for half of the increase over No

SLA requiring twice as much data (one data growth step in privacy-adaptive training). Thus, privacy-adaptive training increases reliability of DP training pipelines for reasonable increase in sample complexity.

Benefit of Block Composition (Q3)

Block composition lets us combine multiple blocks into a dataset, such that each DP query runs over all used blocks with only one noise draw. Without block composition a DP query is split into multiple queries, each operating on a single block, and receiving independent noise. The combined results are more noisy. Figure 2.10(a) and 2.10(c) show the model quality of the LR and NN models on the Taxi dataset, when operating on blocks of different sizes, 100K and 500K for the LR, and 5M for the NN. Figure 2.10(b) and 2.10(d) show the SLAed validation sample complexity of the same models. We compare these configurations against Sage's combined-block training that allows ML training and validation to operate on their full relevance windows. We can see that block composition helps both the training and validation stages. While LR training (Figure 2.10(a)) performs nearly identically for Sage and block sizes of 100K or 500K (6h of data to a bit more than a day), validation is significantly impacted. The LR cannot be validated with any MSE better than 0.003 with block sizes of 500K, and 0.0044 for blocks of size 100K. Additionally, those targets that can be validated require significantly more data without Sage's block composition: 10x for blocks of size 500K, and almost 100x for blocks of 100K. The NN is more affected at training time. With blocks smaller than 1M points, it cannot even be trained. Even with an enormous block size of 5M, more than ten days of data (Figure 2.10(c)), the partitioned model performs 8% worse than with Sage's block composition. Although on such large blocks validation itself is not much affected, the worse performance means that models can be validated up to an MSE target of 0.0025 (against Sage's 0.0023), and requires twice as much data as with block composition.

Multi-pipeline Workload Performance (Q4)

Last is an end-to-end evaluation of Sage with a workload consisting of a data stream and ML pipelines arriving over discrete time steps. At each step, a number of new data points corresponding approximately to 1 hour of data arrives (16K for Taxi, 267K for Criteo). The time between new pipelines is drawn from a Gamma distribution. When a new pipeline arrives, its sample complexity (number of data points required to reach the target) is drawn from a power law distribution, and a pipeline with the relevant sample complexity is chosen uniformly among our configurations and targets (Table 2.1). Under this workload, we compare the average model release in steady state for four different strategies. This first two leverage *Query Composition* and *Streaming Composition* from prior work, as explained in methodology and § 2.3. The other two take advantage of Sage's Block Composition. Both strategies uniformly divide the privacy budget of new blocks among all incomplete pipelines, but differ in how each pipeline uses its budget. *Block/Aggressive* uses as much privacy budget as is available when a pipeline is invoked. *Block/Conserve (Sage)* uses the Privacy-Adaptive Training strategy defined in § 2.3.

Figure 2.11 shows each strategy's average model release time under increasing load (higher model arrival rate), as the system enforces $(\epsilon_g, \delta_g) = (1.0, 10^{-6})$-DP over the entire stream. We make two observations. First, Sage's block composition is crucial. Query Composition and Streaming Composition quickly degrade to off-the-charts release times: supporting more than one model every two hours is not possible and yields release times above three days. On the other hand, strategies leveraging Sage's block composition both provide lower release times, and can support up to 0.7 model arrivals per hour (more than 15 new models per day) and release them within a day. Second, we observe consistently lower release times under the privacy budget conserving strategy. At higher rates, such as 0.7 new models per hour, the difference starts to grow: Block/Conserve has a release time 4x and 2x smaller than Block/Aggressive on Taxi (Figure 2.11(a)) and Criteo (Figure 2.11(b)) respec-

tively. Privacy budget conservation reduces the amount of budget consumed by an individual pipeline, thus allowing new pipelines to use the remaining budget when they arrive.

2.5 Discussion and Analysis

As companies disseminate ML models trained over sensitive data to untrusted domains, it is crucial to start controlling data leakage through these models. We presented *Sage*, the first ML platform that enforces a global DP guarantee across all models released from sensitive data streams. Its main contributions are its *block-level accounting* that permits endless operation on streams and its *privacy-adaptive training* that lets developers control DP model quality. The key enabler of both techniques is our systems focus on ML training workloads rather than DP ML's typical focus on individual training algorithms. While individual algorithms see either a static dataset or an online training regime, workloads interact with *growing databases*. Across executions of multiple algorithms, new data becomes available (helping to renew privacy budgets and allow endless operation) and old data is reused (allowing training of models on increasingly large datasets to appease the effect of DP noise on model quality).

We believe that this systems perspective on DP ML presents further opportunities worth pursuing in the future. Chief among them is how to allocate data, privacy parameters, and compute resources to conserve privacy budget while training models efficiently to their quality targets. Sage proposes a specific heuristic for allocating the first two resources (§2.3), but leaves unexplored tradeoffs between data and compute resources. To conserve budgets, we use as much data as is available in the database when a model is invoked, with the lowest privacy budget. This gives us the best utilization of the privacy resource. But training on more data consumes more compute resources. Identifying principled approaches to perform these allocations is an open problem.

A key limitation of this work is its focus on event-level privacy, a semantic that is insufficient when groups of correlated observations can reveal sensitive information. The best

known example of such correlation happens when a user contributes multiple observations, but other examples include repeated measurements of a phenomenon over time, or users and their friends on a social network. In such cases, observations are all correlated and can reveal sensitive information, such as a user's demographic attributes, despite event-level DP. It should be noted that even in the face of correlated data DP holds for each individual observation: other correlated observations constitute side information, to which DP is known to be resilient. Still, to increase protection, an exciting area of future work is to add support for and evaluate user-level privacy. Our block accounting theory is amenable to this semantic (§2.3), but finding settings where the semantic can be sustained without running out of budget is an open challenge.

2.6 Related Work

Sage's main contribution – block composition – is related to *DP composition theory*. Basic[53] and strong[58;90] composition theorems give the DP guarantee for multiple queries with adaptively chosen computation. McSherry[111] and Zhang, et.al.[183] show that non-adaptive queries over non-overlapping subsets of data can share the DP budget. Rogers, et.al.[136] analyze composition under adaptive DP parameters, which is crucial to our block composition. These works all consider fixed datasets and query-level accounting.

Compared to all these works, our main contribution is to formalize the new block-level DP interaction model, which supports ML workloads on growing databases while enforcing a global DP semantic without running out of budget. This model sits between traditional DP interaction with static data, and streaming DP working only on current data. In proving our interaction model DP we leverage prior theoretical results and analysis methods. However, the most comprehensive prior interaction model[136] did not support all our requirements, such as interactions with adaptively chosen data subsets, or future data being impacted by previous queries.

Streaming DP[31;52;54;55] extends DP to data streams but is restrictive for ML. Data is consumed once and assumed to never be used again. This enables stronger guarantees, as data need not even be kept internally. However, training ML models often requires multiple passes over the data.

Cummings, et.al.[45] consider *DP over growing databases*. They focus on theoretical analysis and study two setups. In the fist setup, they also run DP workloads on exponentially growing data sizes. However, their approach only supports linear queries, with a runtime exponential in the data dimension and hence impractical. In the second setup, they focus on training a single convex ML model and show that it can use new data to keep improving. Supporting ML workloads would require splitting the privacy budget for the whole stream among models, creating a running out of privacy budget challenge.

A few *DP systems* exist, but none focuses on streams or ML. PINQ[111] and its generalization wPINQ[131] give a SQL-like interface to perform DP queries. They introduce the partition operator allowing *parallel composition*, which resembles Sage's block composition. However, this operator only supports non-adaptive parallel computations on non-overlapping partitions, which is insufficient for ML. Airavat[139] provides a MapReduce interface and supports a strong threat model against actively malicious developers. They adopt a perspective similar to ours, integrating DP with access control. GUPT[116] supports automatic privacy budget allocation and lets programmers specify accuracy targets for arbitrary DP programs with a real-valued output; it is hence applicable to computing summary statistics but not to training ML models. All these works focus on static datasets and adopt a generic, query-level accounting approach that applies to any workload. Query-level accounting would force them to run out of privacy budget if unused data were available. Block-level accounting avoids this limitation but applies to workloads with specific data interaction characteristics (§2.3).

(a) QueryCompose($\mathcal{A}$, b, r, $(\epsilon_i, \delta_i)_{i=1}^r$):

for i in $1, \ldots, r$ **do** ▷ ($\mathcal{A}$ depends on $V_1^b, \ldots, V_{i-1}^b$ in iter. i)
  $\mathcal{A}$ gives neighboring datasets $\mathcal{D}^{i,0}$ & $\mathcal{D}^{i,1}$
  $\mathcal{A}$ gives (ϵ_i, δ_i)-DP $\mathcal{Q}_i$
  $\mathcal{A}$ receives $V_i^b = \mathcal{Q}_i(\mathcal{D}^{i,b})$
end forreturn $V^b = (V_1^b, \ldots, V_r^b)$

Figure 2.4: Traditional Query-level Accounting.

(b) BlockCompose($\mathcal{A}$, b, r, $(\epsilon_i, \delta_i)_{i=1}^r$, $(\text{blocks}_i)_{i=1}^r$):

$\mathcal{A}$ gives two neighboring block datasets $\mathcal{D}^0$ and $\mathcal{D}^1$
for i in $1, \ldots, r$ **do** ▷ ($\mathcal{A}$ depends on $V_1^b, \ldots, V_{i-1}^b$ in iter. i)
  $\mathcal{A}$ gives (ϵ_i, δ_i)-DP $\mathcal{Q}_i$
  $\mathcal{A}$ receives $V_i^b = \mathcal{Q}_i(\bigcup_{j \in \text{blocks}_i} \mathcal{D}_j^b)$
end forreturn $V^b = (V_1^b, \ldots, V_r^b)$

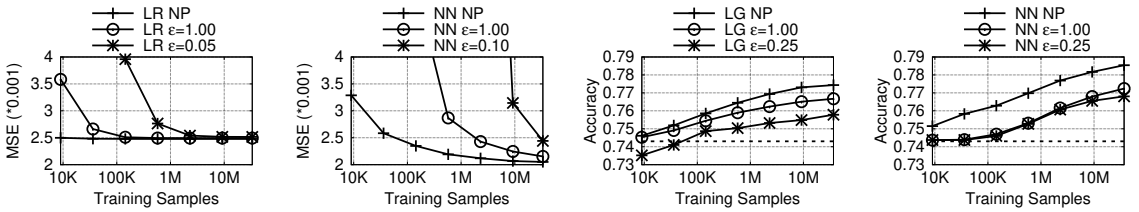
Figure 2.5: Block Composition for Static Datasets. Change from query-level accounting shown in yellow background.

(c) AdaptiveStreamBlockCompose($\mathcal{A}$, b, r, ϵ_g, δ_g, $\mathcal{W}$):

1: $\mathcal{A}$ gives k, the index of the block with the adversarially chosen change
2: **for** i in $1, \ldots, r$ **do** ▷ ($\mathcal{A}$ depends on $V_1^b, \ldots, V_{i-1}^b$ in iter. i)
3: **if** create new block l and $l == k$ **then**
4: $\mathcal{A}$ gives neighboring blocks $\mathcal{D}_k^0$ and $\mathcal{D}_k^1$
5: **else if** create new block l and $l \neq k$ **then**
6: $\mathcal{D}_l^b = \mathcal{D}(\mathcal{W}, V_1^b, \ldots, V_{i-1}^b)$
7: **end if**
8: $\mathcal{A}$ gives blocks_i, (ϵ_i, δ_i), and (ϵ_i, δ_i)-DP $\mathcal{Q}_i$
9: **if** $\bigwedge_{j \in \text{blocks}_i} \text{AccessControl}^j_{\epsilon_g, \delta_g}(\epsilon_1^j, \delta_1^j, ..., \epsilon_i^j, \delta_i^j, 0, ...)$ **then**
10: $\mathcal{A}$ receives $V_i^b = \mathcal{Q}_i(\bigcup_{j \in \text{blocks}_i} \mathcal{D}_j^b)$
11: **else**$\mathcal{A}$ receives no-op $V_i^b = \perp$
12: **end if**
13: **end forreturn** $V^b = (V_1^b, \ldots, V_r^b)$

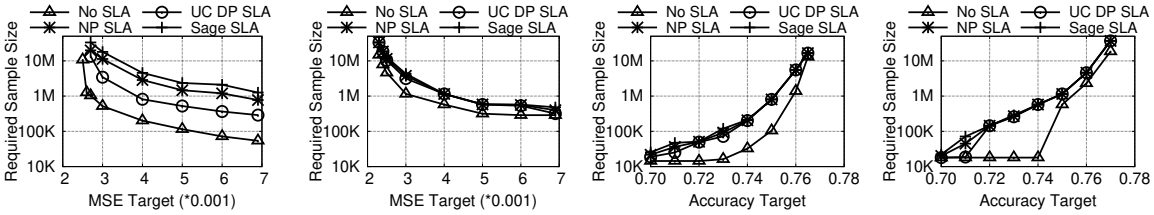
Figure 2.6: Sage Block Composition. Adds support for streams (yellow lines 1-6) and adaptive choice of blocks, privacy parameters (green lines 7-8).

Figure 2.7: Interaction Protocols for Composition Analysis. $\mathcal{A}$ is an algorithm defining the adversary's power; $b \in \{0, 1\}$ denotes two hypotheses the adversary aims to distinguish; r is the number of rounds; $(\epsilon_i, \delta_i)_{i=1}^r$ the DP parameters used at each round; $(\text{blocks}_i)_{i=1}^r$ the blocks used at each round. $\text{AccessControl}^j_{\epsilon_g, \delta_g}$ returns true if running (ϵ_i, δ_i)-DP query $\mathcal{Q}_i$ on block j ensures that with probability $\geq (1 - \delta_g)$ the privacy loss for block j is $\leq \epsilon_g$.



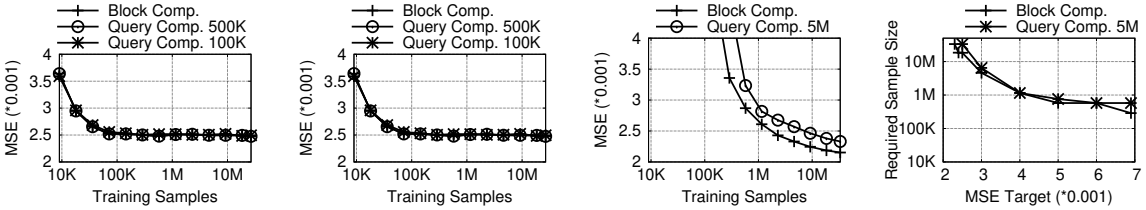
(a) **Taxi LR MSE** (b) **Taxi NN MSE** (c) **Criteo LG Accuracy** (d) **Criteo NN Accuracy**

Figure 2.8: Impacts on TFX Training Pipelines. Impact of DP on the overall performance of training pipelines. 2.9(a), and 2.8(b) show the MSE loss on the Taxi regression task (lower is better). 2.8(c); 2.8(d) show the accuracy on the Criteo classification task (higher is better). The dotted lines are naïve model performance.



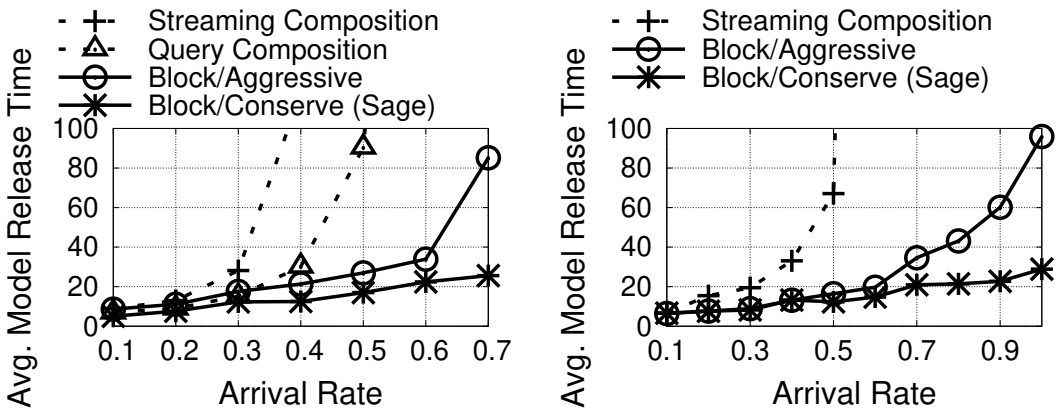
(a) **Taxi LR ACCEPT** (b) **Taxi NN ACCEPT** (c) **Criteo LG ACCEPT** (d) **Criteo NN ACCEPT**

Figure 2.9: Number of Samples Required to ACCEPT models at achievable quality targets. For MSE targets (Taxi regression 2.9(a), and 2.9(b)) small targets are harder to achieve and require more samples. For accuracy targets (Criteo classification 2.9(c), and 2.9(d)) high targets are harder and require more samples. the number of samples to ACCEPT models at achievable MSE targets on the Taxi regression task.



(a) **Taxi LR MSE** (b) **Taxi LR ACCEPT** (c) **Taxi DNN MSE** (d) **Taxi DNN ACCEPT**

Figure 2.10: Block-level vs. Query-level Accounting. Block-level query accounting provides benefits to model quality and validation.



(a) **Taxi Dataset** (b) **Criteo Dataset**

Figure 2.11: Average Model Release Time Under Load.

Chapter 3

Pyramid: Enhancing Selectivity in Big Data Protection with Count Featurization

3.1 Introduction

Driven by cheap storage and the immense perceived potential of "big data," both public and private sectors are accumulating vast quantities of personal data: clicks, locations, visited websites, social interactions, and more. Data offers unique opportunities to improve personal and business effectiveness. It can boost applications' utility by personalizing their features; increase business revenues via targeted product placement; improve social processes such as healthcare, disaster response and crime prevention. Its commercialization potential, whether real or perceived, drives unprecedented efforts to grab and store raw data resources that can later be mined for profit.

Unfortunately, this "collect-everything" mentality poses serious risks for organizations by exposing extensive data stores to external and internal attacks. The hacking and exploiting of sensitive corporate and governmental information have become commonplace[63;77]. Privacy-transgressing employees have been discovered snooping into data stores to spy on friends, family, and job candidates[73;125]. Although organizations strive to restrict access to particularly sensitive data (such as passwords, SSNs, emails, banking data), properly managing access controls for diverse and potentially sensitive information remains an unanswered problem.

Compounding this challenge is a significant new thrust in the public and private spheres to integrate data collected from multiple sources into a single, giant repository (or "data lake") and make that available to any applications that might benefit from it[174;132;38]. This practice magnifies the data exposure problem, transforming big data into what some have called a "toxic asset"[143].

Our goal in this paper is to explore a more rigorous and selective approach to big data protection. We hypothesize that not all data that is collected and archived is, or may ever be, needed or used. The ability to distinguish data needed now or in the future from data collected "just in case" could enable organizations to restrict the latter's exposure to attacks.

For example, one could ship unused data to a tightly controlled store, whose read accesses are carefully mediated and audited. Turning this hypothesis into a reality requires finding ways to: (1) minimize data kept in the company's widely-accessible data lakes, and (2) avoid the need to access the controlled store to meet current and evolving workload needs.

A natural approach might be to monitor data use and retain only the working set of in-use data in accessible storage; data unused for some time is evicted to the protected store[162]. However, many of today's big data applications involve machine learning (ML) workloads that are periodically retrained to incorporate new data, resulting in frequent accesses to all data. How can we determine and minimize the training set—the "working set" for emerging ML workloads—to adopt a more rigorous and selective approach to big data protection?

We observe that for ML workloads, significant research is devoted to limiting the amount of data required for training. The reasons are many but typically do not involve data protection. Rather, they include increasing performance, dealing with sparsity, and limiting labeling effort. Techniques such as dimensionality reduction[28], feature hashing[152], vector quantization[68], and count featurization[158] are routinely applied in practice to reduce data dimensionality so models can be trained on manageable training sets. Semi-supervised[188] and active learning[149] reduce the amount of labeled data needed for training when labeling requires manual effort.

Can such mechanisms also be used to limit exposure of the data being collected? How can an organization that already uses these methods develop a more robust data protection architecture around them? What kinds of protection guarantees can this architecture provide?

As a first step to answering these questions, we present *Pyramid*, a limited-exposure big-data management system built around a specific training set minimization method called *count featurization*[32;36;103;158]. Also called historical statistics, count featurization is a widely used technique for reducing training times by feeding ML algorithms with a limited subset

of the collected data combined (or *featurized*) with historical aggregates from much larger amounts of data. The method is valuable when features with strong predictive power are highly dimensional, requiring large quantities of data (and large amounts of time and resources) to be properly modeled. Applications that use count featurization include targeted advertising, recommender systems, and content personalization systems. Such applications rely on user information to predict clicks, but since there can be hundreds of millions of users, training can be very expensive without some way to aggregate users, like count featurization. The advertising systems at Microsoft, Facebook, and Yahoo are all built upon this mechanism[25], and Microsoft Azure offers it as a service[18].

Pyramid builds on count featurization to construct a selective data protection architecture that minimizes exposure of individual observations (e.g., individual clicks). To highlight, Pyramid: keeps a small, rolling window of accessible raw data (the *hot window*); summarizes the history with privacy-preserving aggregates (called *counts*); trains application models with hot raw data featurized with counts; and rolls over the counts to forget all traces of observations past a specified retention period. Counts are infused with differentially private noise[53] to protect individual observations that are no longer in the hot window but still fall within the retention period. Counts can support modifications and additions of many (but not all) types of models; historical raw data, which may be needed for workloads not supported by count featurization, is kept in an encrypted store whose decryption requires special access.

While count featurization is not new, our paper is the first to retrofit it for data protection. Doing so raises significant challenges. We first need to define meaningful requirements and protection guarantees that can be achieved with this mechanism, such as the amount of exposed information or the granularity of protection. We then need to achieve these protection guarantees without affecting model accuracy and scalability, despite using much less raw data. Finally, to make the historical raw data store easier to protect, we need to access it as little as possible. This means supporting workload evolution, such as parameter tuning or

trying new algorithms, without the need to go back to historical raw data store.

We overcome these challenges with three main techniques: (1) *weighted noise infusion*, which automatically shares the privacy budget to give noise-sensitive features less noise; (2) an *unbiased private count-median sketch*, a data structure akin to a count-min sketch that resolves the large negative bias arising from applying differentially private noise to a count-min sketch; and (3) *automatic count selection*, which detects potentially useful groups of features to count together, to avoid accesses to the historical data. Together, these techniques reduce the impact of differentially private noise and count featurization.

We built Pyramid and integrated it into Spark Velox, a targeting and personalization framework, to add rigor and selectivity to its data management. We evaluated three applications: a targeted advertising system using the Criteo dataset, a movie recommender using the MovieLens dataset, and MSN's production news personalization system. Results show that: (1) Pyramid approaches state-of-the-art models while training on less than *1% of the raw data*. (2) Protecting historical counts with differential privacy has only *2% impact on accuracy*. (3) Pyramid adds just *5% performance overhead*.

Overall, we make the following contributions:

1. Formulating the *selective data protection problem* for emerging ML workloads as a training set minimization problem, for which many mechanisms already exist.
2. The design of Pyramid, the first selective data management system that minimizes data exposure in anticipation of attack. Built upon count featurization, Pyramid is particularly suited for targeting and personalization workloads.
3. A set of new techniques to balance solid protection guarantees with model accuracy and scalability, such as our unbiased private count-median sketches.

   library ready to integrate in other targeting/personalization frameworks. `https://columbia.github.io/selective-data-systems/`

3.2 Motivation and Goals

This paper argues for needs-based selectivity in big data protection: protecting data differently depending on whether or not it is actually needed to handle a company's day-to-day workloads. Intuitively, data that is needed day-to-day is less amenable to certain kinds of protection (e.g., auditing or case-by-case access control) than data needed only for exceptional situations. A key question is *whether a company's day-to-day needs can be captured with a limited and well-defined data subset*. While we do not claim to answer this question in full, we present with Pyramid the first evidence that selectivity can be achieved in one important big-data workload domain: *ML-based targeting and personalization*. The following scenario motivates selectivity and shows how and in what contexts Pyramid helps improve protection.

Example Use Case

MediaCo, a media conglomerate, collects *observations* of user behavior from its hundreds of affiliate news and entertainment sites. Observations include the articles users read and share, the ads they click, and how they respond to A/B testing. MediaCo uses this data to optimize various processes, including recommending articles to users, showing the most relevant articles first, and targeting ads. Initially, MediaCo collected observations from affiliate sites in separate, isolated repositories; different engineering teams used different repos to optimize these processes for each affiliate site. Recently, MediaCo has started to track users across sites using cookies and to integrate all data into a central data lake. Excited about the potential of the much richer information in the data lake, MediaCo plans to provide indiscriminate access to all engineers. However, aware of recent external hacking and insider attacks affecting other companies, it worries about the risks it assumes with such wide access.

MediaCo decides to use Pyramid to limit the exposure of historical observations in anticipation of such attacks. For MediaCo's *main workloads*, which consist of targeting and

personalization, the company already uses count featurization to address sparsity challenges; hence, Pyramid is directly applicable for those workloads. They configure it by keeping Pyramid's hot window of raw observations, along with its noise-infused historical statistics, in the widely accessible data lake so all engineers can train their models, tune them, and explore new algorithms every day. Pyramid absorbs many workload needs—current and evolving—as long as the algorithms draw on the same user data to predict the same outcome (e.g., whether a user will click on an ad). MediaCo also configures a one-year retention period for all observations; after this period, Pyramid removes observations from the statistics and launches retraining of all application models to purge the old activity. Finally, MediaCo stores all raw observations in an encrypted store whose read accesses are disabled by default. Access to this store is granted temporarily and on a case-by-case basis to engineers who demonstrate the need for statistics beyond those that Pyramid maintains.

In addition to targeting/personalization workloads, MediaCo has *other, potentially non-ML workloads*, such as business analytics, trend studies, and forensics; for these, count featurization may not apply. Hence, MediaCo gives direct access to the raw-data store to engineers managing these workloads and isolates their computational resources from the targeting/personalization teams.

With this configuration, MediaCo minimizes access to its collected data on a needs basis. Assuming no entity with full access to the historical raw data is malicious, Pyramid guarantees the following (detailed in §3.2). (1) Any observations preceding the hot window when an attack begins will be hidden from the attacker. (2) Hiding is done at an individual observation level during the retention period and in bulk past the retention period. (3) Only in exceptional circumstances do engineers get access to the historical raw data. With these guarantees, MediaCo negotiates lower data loss insurance premiums and gains PR benefits for its efforts to protect user data.

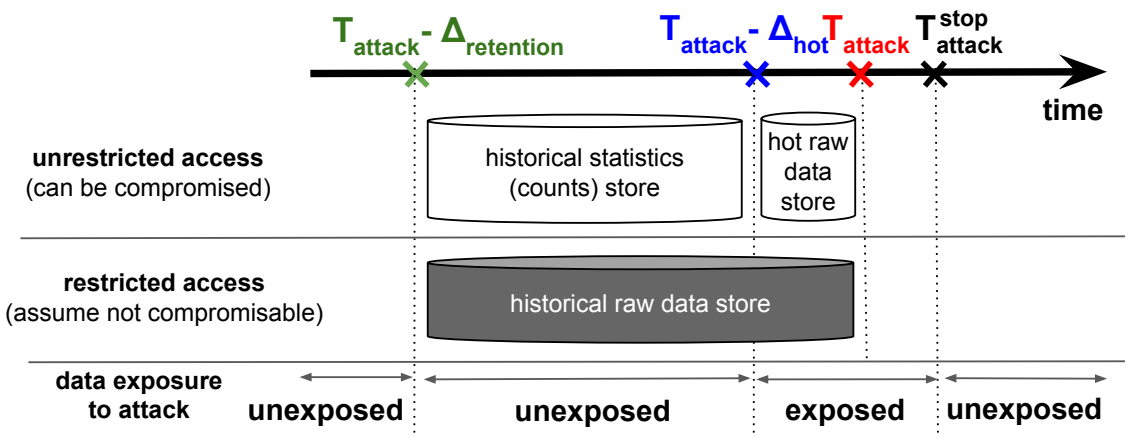


Figure 3.1: Threat model. T_{attack}: time the attack starts; T^{stop}_{attack}: time the attack is eradicated; Δ_{hot}: hot window length; $\Delta_{retention}$: company's data retention period.

Threat Model

Figure 3.1 illustrates Pyramid's threat model and guarantees. Pyramid gives guarantees similar to those of forward secrecy: a one time compromise will not allow an adversary to access all past data. Attacks are assumed to have a well-defined start time, T_{attack}, when the adversary gains access to the machines charged with running Pyramid, and a well-defined end time, T^{stop}_{attack}, when administrators discover and stop the intrusion. Adversaries are assumed to not have had access to the system before T_{attack}, nor to have performed any action in anticipation of their attack (e.g., monitoring external predictions, the hot window, or the models' state), nor to have continued access after T^{stop}_{attack}. The attacker's goal is to exfiltrate individual observations of user activities (e.g., to know if a user clicked on a specific article/ad). Historical raw data is assumed to be protected through independent means and not compromised in this attack. Pyramid's goal is to limit the hot data in active use, which is widely accessible to the attacker.

Examples of adversaries that fit our threat model can be found among both the internal and external adversaries of a company. An external adversary may be a hacker who breaks into the company's computing infrastructure at time T_{attack} and starts looking for data that may prove of value (e.g., information about celebrities' specific activities, what they liked or disliked, where they were in the past, etc.). An internal adversary may be a privacy-

transgressing employee who *spontaneously* decides at T_{attack} to look into some past action of a family member or friend (e.g., to check if the person has visited or liked a particular page).

After compromising Pyramid's internal state, the attacker will gain access to data in three different representations: the hot data store containing plaintext observations, the historical counts, and the trained models themselves. The plaintext observations in the hot data store are not protected in any way. The historical statistics store contains differentially private count tables of the recent past. The attacker will learn some information from the count tables but individual records will be protected with a differentially private guarantee. Pyramid forces models to be retrained when observations are removed from the hot raw data store, so the attacker will not be able to learn anything from the models beyond what they have already learned above.

Pyramid provides three protection levels:

P1 *No protection for present or future observations.* Observations in the hot data store when the attack begins, plus observations added to the hot data store while the attack is ongoing, receive no protection; i.e., observations received between ($T_{attack} - \Delta_{hot}$) and T_{attack}^{stop} receive no protection.

P2 *Protection for individual observations for the length of the retention period.* Statistics about observations are retained in differentially private count tables for a predefined retention period $\Delta_{retention}$. The attacker may learn broad statistics about observations in the interval $[T_{attack} - \Delta_{retention}, T_{attack} - \Delta_{hot}]$ but will not be able to confidently determine if a specific observation is present in the table.

P3 *Protection in bulk past the retention period.* Observations past their retention period (i.e., older than $T_{attack} - \Delta_{retention}$) have been phased out of the historical statistics store and are protected separately by the historical raw data store.

Finally, we assume that no states created based on the hot raw data persist once the hot window is rolled over. While we explicitly launch retraining of models registered with Pyra-

mid, we operate under the assumption that (1) the models' states are securely erased[79] and (2) no other state was created out of band based on the raw hot data (such as copies made by programmers).

Design Requirements

Given the threat model, our design requirements are:

R1 *Limit widely accessible data.* The hot data window is exposed to attackers; hence, Pyramid must limit its size subject to application-level requirements, such as the accuracy of models trained with it.

R2 *Avoid accesses to historical raw data even for evolving workloads.* Pyramid must absorb as many current and evolving workload needs as possible to limit access to the historical raw data.

R3 *Support retention policies.* Pyramid must enforce a company's retention policies. Although Pyramid provides a differential privacy guarantee, no protection is stronger than securely deleting data.

R4 *Limit impact on accuracy, performance, scalability.* We intend to preserve the functional properties of applications and models running on Pyramid.

3.3 The Pyramid Architecture and Design

Pyramid, the first selective data management architecture, builds upon the ML technique of count-based featurization and augments it with new mechanisms to meet the preceding design requirements.

Background on Count-Based Featurization

Training predictive models can be challenging on data that contains categorical variables (features) with large numbers of possible values (e.g., an ID or an interest vector). Existing ML techniques that handle large feature spaces often make strong assumptions about the data, e.g., assuming a linear relationship between the features and the label (e.g., Lasso[165]). If the data does not meet these assumptions, results can be very poor.

Count-based featurization[158] is a popular approach to handling categorical variables of high cardinality. Rather than directly using the value of a categorical variable, this technique featurizes the data with the number of times a particular feature value (e.g., a user ID) was observed with each label and the conditional probability of the label given the feature value. This substantially reduces dimensionality. Suppose the raw data contains d categorical features with an average cardinality of K and a label of cardinality L, where $K \gg L$; e.g., in click prediction K can be millions (number of users), while L is 2 (click, non-click). Standard encoding of categorical variables[14] results in a feature space of dimension $O(dK)$, whereas with count featurization it is $O(dL)$. Count featurization can also be applied to continuous variables or continuous labels by first discretizing them; this increases dimensionality but only by a small factor.

The dramatic dimensionality reduction yields important benefits. It is known that fewer dimensions permit more efficient learning, both statistically and computationally, potentially at the cost of reducing predictive accuracy. However, count featurization makes it feasible to apply advanced, nonlinear models, such as neural networks, boosted trees, and random forests. This combination of succinct data representation and powerful learning models enables substantial reduction of the training data with little loss in predictive performance. Quantified in §3.5, this is the insight behind our use of count-based featurization to limit data exposure.

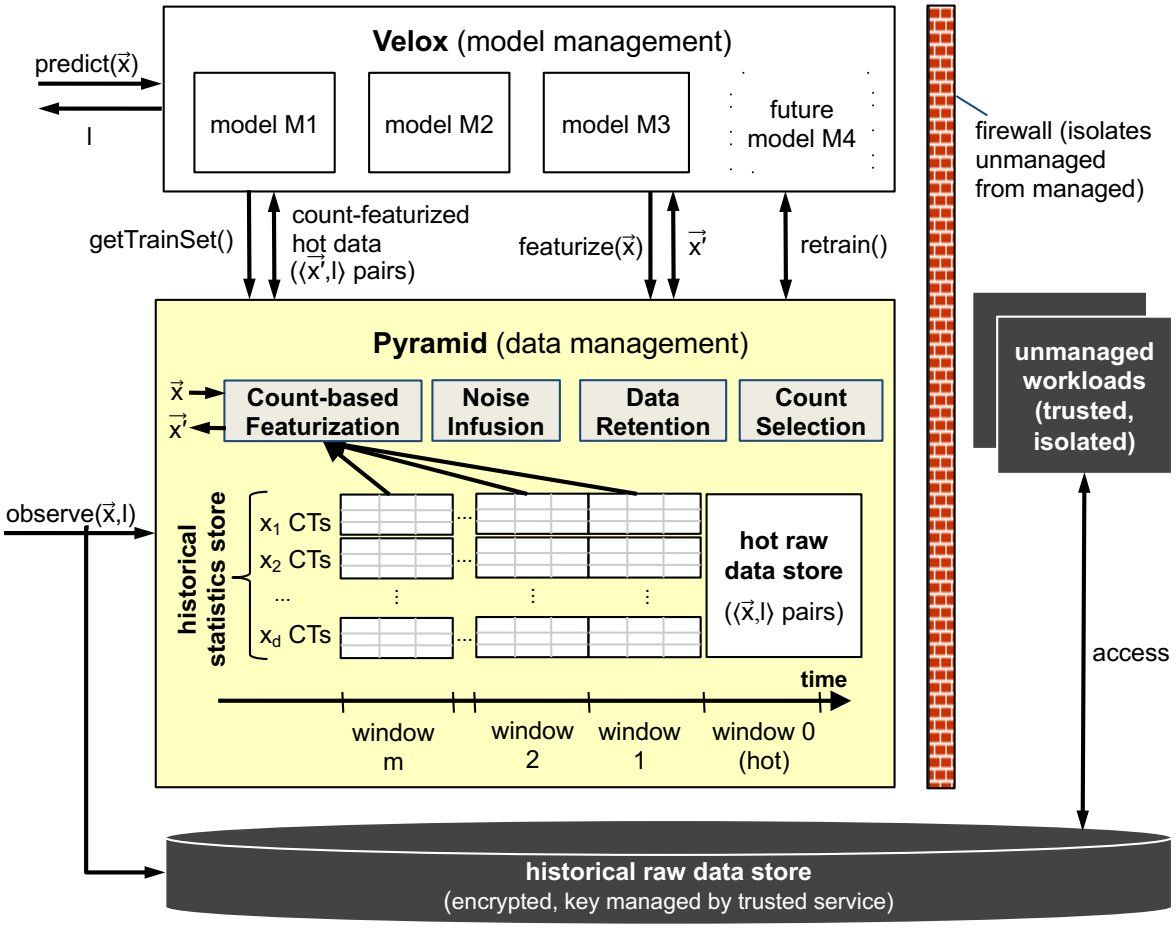


Figure 3.2: Pyramid's architecture. Notation: $\vec{x}$: feature vector; l: label; $\vec{x'}$: count-featurized feature vector; CT: count table.

Architectural Components

Figure 3.2 shows Pyramid's architecture. Pyramid manages collected data (observations) on behalf of application models hosted by a model management system. In our case, we use Velox[44], built on Spark. Velox facilitates ML-based targeting and personalization services by implementing three functions: (1) fast, but incomplete, incorporation of new observations into models that programmers register with Velox; (2) low-latency prediction serving from these models; and (3) periodic retraining of the models to correct inconsistencies created by the incomplete incorporation of new observations. Velox saves observations in a separate data management component, Spark's Tachyon. Pyramid replaces this component to ensure rigorous and selective protection of observations.

Pyramid itself consists of four architectural components, shown across the top of the highlighted box in Figure 3.2. The first is *count featurization*, which leverages the known ML mechanism to count featurize observations before feeding them to models for training and prediction. The second, third, and fourth are *noise infusion*, *data retention*, and *count selection*, which augment count featurization with differential privacy and a set of new mechanisms to meet Pyramid's design requirements. We discuss each component in turn.

Count Featurization

Pyramid hijacks the stream of observations collected by Velox (the `observe` method) and count-featurizes them. An observation is a pair $\langle \vec{x}, l \rangle$ with a feature vector $\vec{x} = \langle x_1, x_2, ..., x_d \rangle$ and a label l. Application models predict the label (or a probability for each possible label) for a given feature vector by training on count-featurized observations. When an observation arrives, Pyramid incorporates it into two data structures: (1) the *hot raw data store*, which retains observations from the recent past, and (2) the *historical statistics store*, which consists of multiple *count tables* that maintain the number of occurrences of each feature with each label. We maintain count tables for all features in $\vec{x}$ and for some feature combinations. A separate set of count tables is maintained for each time window.

Featurization transforms a feature vector $\vec{x}$ into a count-featurized feature vector $\vec{x'}$, by replacing each feature x_i with the conditional probabilities of each label value given x_i's value. The conditional probabilities are computed directly from the count tables as discussed below. To train its models, an application requests a training set from Pyramid (`getTrainSet`). Pyramid featurizes the hot raw data with historical counts and returns it to the application. To predict the label for a feature vector $\vec{x}$, the application requests its featurization from Pyramid (`featurize`); Pyramid returns $\vec{x'}$.

Example. Figure 3.3 shows (a) a sample observation format, (b) some count tables used by Pyramid to count-featurize it, and (c) a sample count-featurized observation.

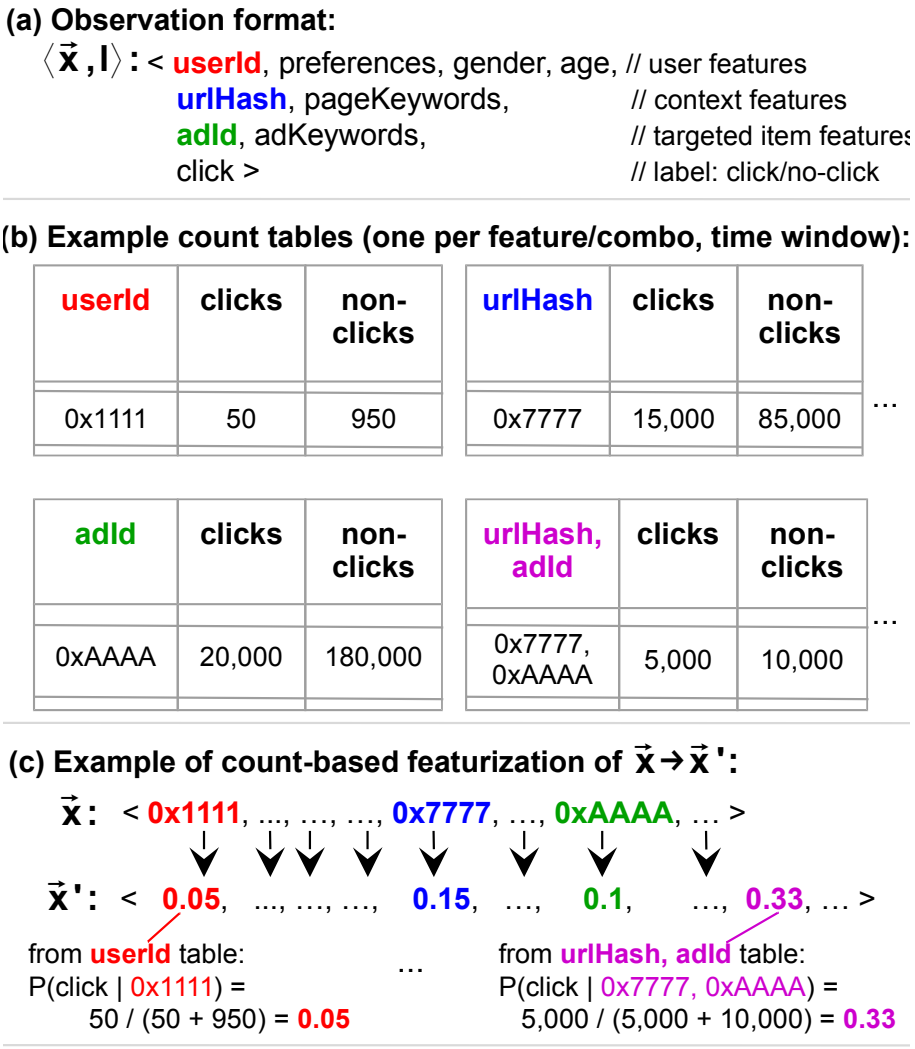


Figure 3.3: Count featurization example.

• *Observation format.* In targeting and personalization, an observation's feature vector $\vec{x}$ typically consists of *user features* (e.g., id, gender, age, and previously compiled preferences) and *contextual information* for the observation (e.g., the URL of the article or the ad shown to the user, plus any features of these). The label l might indicate whether the user clicked on the article/ad.

• *Count tables.* Once an observation stream of the preceding type is registered with Pyramid, the *userId* table maintains for each user the number of clicks the user has made on any ad shown and the number of non-clicks; it therefore encodes each user's propensity to click on ads. The *urlHash* table maintains for each URL the number of clicks that each user made

on any ad shown on that page; it therefore encodes the page's inherent "ad-clickability." Pyramid maintains count tables for every feature in $\vec{x}$ and for some feature combinations with predictive potential, such as the $\langle \textit{urlHash}, \textit{adId} \rangle$ table, which encodes the joint probability of a particular ad being clicked when it is shown on a particular page.

• *Count featurization.* To count-featurize a feature vector $\vec{x} = \langle x_1, x_2, \ldots, x_d \rangle$, Pyramid first replaces each of its features with the conditional probabilities computed from the count tables, e.g., $\vec{x'} = \langle P(\textit{click}|x_1), P(\textit{click}|x_2), \ldots, P(\textit{click}|x_d) \rangle$, where $P(\textit{click}|x_i) = \frac{\textit{clicks}}{\textit{clicks+non-clicks}}$ from the row matching the value of x_i in the table corresponding to x_i. Pyramid also appends to $\vec{x'}$ the conditional probabilities for any feature combinations it maintains. Figure 3.3(c) shows an example of feature vector $\vec{x}$ and its count-featurized version $\vec{x'}$. This is a simplified version of the count featurization function. We can also include the raw counts in $\vec{x'}$, and support non-binary categorical labels by including conditional probabilities for each label. To avoid featurizing with an effectively random probability when a given feature value has very few counts, we estimate the variance of our probability estimate and, if it is too high, featurize with a default probability $P(click)$.

• *Training and prediction.* Suppose a boosted-tree model is trained on a count-featurized dataset ($\langle \vec{x'}, l \rangle$ pairs). It might find that for users with a click propensity over 0.04, the chances of a click are high for ads whose clickability exceeds 0.05 placed on websites with ad-clickability over 0.1. In this case, the model would predict a "click" label for the feature vector in Figure 3.3(c).

Process. Pyramid count-featurizes all features x_i for each observation type. For categorical features, we featurize them as described above. For low-cardinality features, we can additionally include the raw feature values in $\vec{x'}$ alongside the conditional probabilities. Continuous features are first mapped to a discrete space, binning them by percentiles, and then count-featurized as categorical. We do the same with continuous labels.

Pyramid maintains hot windows and count tables as follows. There is one hot window

for each observation stream. There is one count table per feature or feature group; it has a column for each label and a row for each value the feature can take. To support granular retention times, each count table is composed of multiple windowed count tables holding data for observations collected during disjoint windows of time. The complete count table is the sum of the associated windowed count tables. When a new observation arrives, it is added to the hot window and made immediately available to the models for (re)training. The hot window is a sliding window that may be sized differently from the count table window. It is also added to the current windowed count table; this count table is withheld when computing the complete count table until it is finished populating. At this point, Pyramid begins using it as part of the featurization process, phases out the oldest count table if it is past its retention period, and begins populating a new count table that has been initialized with differentially private noise. Once count tables are incorporated into the featurization process, they are never updated again.

Count-min sketches (CMSes). A key challenge with count featurization is its storage requirement. For a categorical variable of cardinality K and a label of cardinality L, the count table is of size $O(LK)$. A common solution, used in Azure[18], is to store each table in a Count-Min Sketch (CMS)[41], a data structure that approximates counts in sub-linear space. A CMS consists of a 2D array with an independent hash function for each row. When a new feature arrives, the CMS uses the hash function for each row to assign the feature to a column and increment the value in that cell.

We query the CMS for a feature count by hashing the feature into a column of each row and taking the minimum value. Despite overcounting from collisions, CMS provides sufficiently accurate count estimates to train ML models. With a CMS, we can maintain more and/or larger count tables with bounded storage overheads. This gives developers flexibility in the types of modeling they can do atop in-use data without tapping into the historical data store. The CMS poses challenges to our noise infusion process, as described next.

Noise Infusion

Pyramid's key contribution is to retrofit count featurization, a technique developed for performance and scalability, to protect past observations against exposure to attack. Pyramid infuses noise into the count tables to protect these observations. While we leverage differential privacy methods[53], correctly applying these methods in our context poses scaling challenges. For example, each observation contributes to multiple count tables, increasing the noise required to guarantee differential privacy, and a naïve application degrades accuracy when there are many count tables. We present two techniques to address this challenge. First, we use a *weighted noise infusion* technique to mitigate the impact of noise, allowing us to navigate the privacy/utility trade-off. Second, for high noise levels, we replace the CMS by a count-median sketch[33], a data structure with weaker accuracy guarantees than CMS but that provides an unbiased frequency estimate, making it more robust to negative noise values. To our knowledge, we are the first to observe that the count-median sketch structure is better suited to differential privacy. After a brief overview of differential privacy, we describe these techniques.

Differential privacy properties. Pyramid's noise infusion component uses four differential privacy properties:

1. Privacy guarantees: Let D_1 be the database of past observations, D_2 be a database that differs from D_1 by exactly one observation (i.e., D_2 adds or removes 1 observation), and S the range of all possible count tables that can result from a randomized query $Q()$ that builds a count table from a window of observations. The count table query $Q()$ is ϵ-*differentially private* if $P[Q(D_1) \in S] \leq e^{\epsilon} \times P[Q(D_2) \in S]$. In other words, adding or removing an observation in D_1 does not significantly change the probability distribution of possible count tables; therefore, the count table does not leak significant information about any specific observation[53]. ϵ is called the query's *privacy budget*.

2. Laplace distribution: Let a query's *sensitivity* be the magnitude of the change in the query result triggered by adding or removing a single observation. If the query has sensitivity Δ, then adding noise drawn from a Laplace distribution with scale parameter $\frac{\Delta}{\epsilon}$ guarantees that the result is ϵ-differentially private[53]. Increasing $\frac{\Delta}{\epsilon}$ increases the standard deviation of the distribution (stdev of a Laplace distribution with parameter b is $b\sqrt{2}$).

3. Composability: Differentially private queries are composable: the sum of n ϵ_n-differentially private queries is $(\Sigma\epsilon_n)$-differentially private[111]. This lets us maintain multiple count tables, possibly with different budgets, and combine them without breaking guarantees. (Advanced composition theorems allow sublinear loss in the privacy budget by relaxing the guarantees to (ϵ, δ)-differential privacy[56], but we do not explore that here.)

4. Post-processing resilience: Any computation on a differentially private data release remains differentially private[56]. This is a crucial point for Pyramid's protection guarantees: it ensures that guarantee **P2**, the protection of individual past observations during their lifetime, holds for each model's internal state and outputs. As long as models comply with retrain calls and erase all internal state when they do, their output is differentially private with regard to observations outside the hot window.

Basic noise infusion process. We apply these known properties when creating count tables for the hot window. Upon creating a count table, we initialize each cell of the CMS storing that table with a random draw from a Laplace distribution. This noise is added only once: the count tables are updated as observations arrive and are sealed when the hot window rolls over. To determine the correct parameter for the Laplace distribution, b, we must account for three factors: (1) the internal structure of the CMS, (2) the number of observations we want to hide simultaneously, and (3) the number of count tables (features or feature combinations) we are maintaining.

First, an exact count table has sensitivity 1 since adding or removing an observation can only change one count by 1. For a CMS, each observation is counted once per hash function;

hence, the sensitivity is h, the number of hash functions. Second, if we aim to hide any group of k observations with a privacy budget of ϵ, then we make a count table ϵ-differentially private by adding noise from a Laplace distribution of parameter $b = \frac{hk}{\epsilon}$ in every cell of the CMS. Third, we must maintain multiple count tables for the different features and feature groups. Since each observation affects every count table, we need to split the privacy budget ϵ among them, e.g., splitting it evenly by adding noise with $b = \frac{nhk}{\epsilon}$ to each table.

The third consideration poses a significant challenge for Pyramid: the amount of noise we apply grows linearly with the number of count tables we keep. Since the amount of noise directly affects application accuracy, this yields a protection/accuracy tradeoff, which we address with weighted noise infusion.

Weighted noise infusion process. We note that count tables are not all equally susceptible to noise. For example in our movie recommender, the $user$ table most likely contains low values, since each user rates only a few movies (29 for the median user). Moreover, we do not expect this count to change significantly when adding more data, since single users will not rate significantly more movies. Each $genre$ table however contains higher values (1M or more), since each genre characterizes multiple movies, each rated by many users. Sharing noise equally between tables would pollute all counts by a standard deviation of 145 ($\epsilon = 1$, $h = 5$, and $k = 1$), a reasonable amount for $genre$s, but devastating for the $user$ feature, which essentially becomes random.

Pyramid's weighted noise infusion distributes the privacy budget unevenly across count tables, adding less noise to low-count features. This way, we retain more utility from those tables, and the composability property of differential privacy preserves our protection guarantees. Each table's share of noise is determined automatically, based on the count values observed in the hot window. Specifically, the user specifies a quantile, and the privacy budget is shared between each feature proportionally to this quantile of its counts. For instance we use the first percentile, so that 99% of the counts for a feature will be less affected by the

noise. Sharing the privacy budget proportionally to the counts is a heuristic that makes the noise's standard-deviation proportional to the typical counts of each feature. This scheme is also independent of the learning algorithm.

Finally, the weight selection process should be made differentially private so the weights computed on a previous hot window do not reveal anything about that window's data at a later time. While our implementation currently does not do this, a design might use a small portion of one window's privacy budget and leverage smooth sensitivity [123] to compute differentially private count percentiles that can be used as feature weights. One could compute each weight as a separate differentially private query, or use the sample-aggregate framework and the center of attention aggregation [123] to compute all the weights in one query.

Section 3.5 shows that weighted noise infusion is vital for providing protection while preserving accuracy at scale: without it, the cost of hiding single observations is a 15% accuracy loss; with it, the loss is less than 5%. We leave the evaluation of incorporating differential privacy into the weight selection method for future work.

Unbiased private count-median sketch. Another factor that degrades performance when adding differentially private noise is the interaction between the noise and the CMS. In the CMS, the final estimate for a count is $\min(h_i(\textit{key}))$ for each row i. The minimum makes sense here since collisions can only increase the counts. The Laplace distribution however is symmetric around zero, so we may add negative noise to the counts. Taking the minimum of multiple draws—each cell is initiated with a random draw from the distribution—thus selects the most extreme negative values, creating a downward bias that can be very large for a small ϵ.

We observe that because the mean of the Laplace distribution is 0, an unbiased estimator would not suffer from this drawback. For tables with large noise, we thus use a count-median sketch [33], which differs in two ways: 1) each row i has another hash function s_i that maps the key to a random sign $s_i(\textit{key}) \in \{+1, -1\}$, with each cell updated with $s_i(\textit{key})h_i(\textit{key})$; 2)

the estimator is the median of all counts multiplied by their sign, instead of the minimum. The signed update means that collisions have an expected impact of zero, since they have an equal chance of being negative or positive, making the cell an unbiased estimate of the true count. The median is a robust estimate that preserves the unbiased property.

Using this count-median sketch reduces the impact of noise, since values from the Laplace distribution are exponentially concentrated around the mean of zero. §3.5 shows that for small ϵ, or a large number of features, it is worth trading the CMS's better guarantees for reduced noise impact with the count-median sketch.

Data Retention

While differential privacy provides a reasonable level of protection for past observations, complete removal of information remains the cleanest, strongest form of protection (design **R3** in §3.2). Pyramid supports data expiration with *windowed count tables*. When an observation arrives, Pyramid updates the count tables for the current count window only. To featurize $\vec{x}$, Pyramid sums the relevant counts across windows. Periodically, it drops the oldest window and invokes retraining of all models in Velox (`retrain` method). Our use of count-based featurization supports such behaviors because retraining is cheap (§3.5), so we can afford to do it frequently.

Count Selection

Pyramid seeks to support workload evolution (model changes/additions, such as future model M4 in Figure 3.2) using only the widely accessible stores without tapping into the historical raw data store. To do so, it uses two approaches. First, it stores the count tables in a very compact representation—the count-median sketches—so it can afford to keep plenty of count tables. Second, it includes an automatic process of *count table selection* that inspects the data to identify *feature combinations* worth counting, whether they are used in the current workloads or not. This technique is useful because count featurization tends to

obscure correlations between features. For example, different users may have different opinions about specific ads. Although that information could be inferred by a learning algorithm from the raw data points, it is not accessible in the count-featurized data unless we explicitly count the joint occurrences of specific users with specific ads, i.e., maintain a table for the $\langle userId, adId \rangle$ group.

We adapted several feature selection techniques[80] to select feature groups and describe one here. *Mutual Information* (MI) is a measure of dependence between two random variables. A common feature selection technique keeps features of high MI with the label. We extend this mechanism for group count selection. Our goal is to identify feature groups that provide more information about the label than individual features. For each feature x_i, we find all other features x_j such that x_i and x_j together exhibit higher MI with the label than x_i alone. From these groups, we select a configurable number with highest MIs. To find promising groups of larger sizes, we apply this process greedily, trying out new features with existing groups. For each selected group, Pyramid creates and maintains a count table.

This exploration of promising groups operates on the *hot window of raw data*. Because the hot raw data is limited, the selection may not be entirely reliable. Therefore, count tables for new groups are added on a "trial basis." As more data accumulates in the counts, Pyramid re-evaluates them by computing the MI metric *on the count tables*. With the increased amount of data, Pyramid can make a more reliable decision regarding which count tables to keep and which to drop. Because count selection—like feature selection—is never perfect, we give engineers an API to specify groups that they know are worth counting from domain knowledge. Finally, like the weight selection process, count selection should be made differentially private so the groups selected in a particular hot window, which are preserved over time, do not leak information about the window's data in the future. We leave this for future work.

Supported Workload Evolution

Count featurization is a model-independent preprocessing step, allowing Pyramid to absorb some common evolutions during an ML application's life cycle without tapping the historical raw data store. §3.5 gives anecdotal evidence of this claim from a production workload. This section reviews the types of workload changes Pyramid currently absorbs.

A developer may want to change four aspects of the model: (1) the algorithm used to train the model (2) hyperparameters for the model or for the underlying optimization algorithm, (3) features used by the model, and (4) the predicted label. Pyramid supports (1) and (2), partially supports (3), and usually does not support (4).

• *Algorithm changes: Supported.* Pyramid allows developers to move between types of models and libraries used to train those models as long as they are using features and labels that are already counted. In our evaluation we experimented with linear models and neural networks in Vowpal Wabbit[96] and gradient boosted trees in scikit-learn[127] using the same count tables.

• *Hyperparameter tuning: Supported.* By far the most common type of model change we encountered, both in our own evaluation and in reports from a production setting, was hyperparameter tuning. For example, a developer may want to change model hyperparameters, such as the number of hidden units in a neural network, or tune parameters of the underlying optimization algorithm, such as the learning rate or an L1/L2 regularization penalty. Changing hyperparameters is independent from the underlying features so is supported by Pyramid.

• *Feature changes: Partially supported.* Pyramid supports making minimal feature changes. A developer may want to perform one of three types of feature changes: adding new features, removing existing features, or adding interactions between existing features. Pyramid trivially supports removing existing features, and lets developers add new features if they are based on existing ones. For example, the developer could not create an $\langle Age, Location \rangle$

feature interaction if the individual features were not already counted together. Introducing new feature combinations or interactions requires creating new count tables. This highlights the importance of count selection to support workload evolution.

• *Label changes: Mostly unsupported.* Changes in predicted labels are not supported except if a new label is a subset of an existing label. For example, a news recommender could not start predicting retention time instead of clicks unless retention time was previously declared as a label. As with features, Pyramid can support label changes when the new label is a subset of an existing one. For example, if a label exists that tracks retention time in time buckets, Pyramid can support new, coarser labels, such as the three classes "0 seconds," "less than a minute," and "more than a minute."

Summary

With these components, Pyramid meets the design requirements noted in §3.2, as follows. **R1:** By enhancing the training set with historical statistics gathered over a longer period of time, we minimize the hot data. **R2:** By automatically identifying combinations of features worth maintaining, we avoid having to access the historical raw data for workloads that use the same observation streams to predict the same label. **R3:** By rolling the count windows and retraining the application models, we support data retention policies, albeit at a coarse level. §3.5 evaluates **R4**: accuracy and performance impact.

3.4 Implementation

Pyramid is implemented in 2600 lines of Scala, as a modular library. It integrates into the feature engineering stage of an ML pipeline, before the actual learning algorithms are invoked. The modular backend allows count tables to be stored locally in memory or in a remote datastore such as Redis or Cassandra.

We integrated Pyramid into the Velox model management system[44] with minimal effort, by adding/modifying around 500 lines of code. The changes we made to Velox involve interposing on all of Velox's interfaces that interact with raw data (e.g., adding observations, making predictions, and retraining). Now prediction requests are passed through the Pyramid featurization layer, which performs count featurization.

One of Velox's key contributions is performing low latency predictions by pushing models to application servers. To enable low-latency predictions, Pyramid periodically replicates snapshots of the central count tables to the application servers, allowing them to perform featurization locally. §3.5 evaluates prediction performance in Velox/Pyramid with and without this optimization.

3.5 Evaluation

We evaluate Pyramid using different versions of three data-driven applications: two ad targeting applications, two movie recommendation applications, and MSN's production news personalization system. We compare models on count-featurized data to state-of-the-art models trained on raw data, and answer these questions:

Q1. Can we accurately learn on less data using counts?

Q2. How does past-data protection impact utility?

Q3. Does counting feature groups improve accuracy?

Q4. How efficient is Pyramid?

Q5. To what problems does Pyramid apply?

Our evaluation yields four findings: (1) On classification problems, count featurization lets models perform within 4% of state-of-the-art models while training on less than *1% of the data*. (2) Count featurization enables powerful nonlinear algorithms, such as neural networks and boosted trees, that would be infeasible due to high-cardinality features. (3) Pro-

App	Dataset	Obs.	Feat.	Baseline
Ad targeting (classification)	Criteo Kaggle [3]	45M	39	neural net in Kaggle [2]
Ad targeting (classification)	Criteo Full [4]	1.2B	39	regularized linear model
Movie recommendation (classification)	MovieLens [83]	22M	21	matrix factorization [96]
Movie recommendation (regression)	MovieLens [83]	22M	21	matrix factorization [96]
News personalization (regression)	MSN.com production	24M	507	contextual bandits [100;49]

Table 3.1: Workloads. Apps and datasets; number of observations and features in each dataset; and baselines used for comparison. All baselines are trained using VW [96].

Dataset	Model	Parameters
Criteo-Kaggle	**B:** neural net (nn)	VW. One 35 nodes hidden layer with tanh activation. LR: 0.15. BP: 25. Passes: 20. Early Terminate: 1.
	logistic regression (log. reg.)	VW. LR: 0.5. BP: 26.
	gradient boosted trees (gbt)	Sklearn. 100 trees with 8 leaves. Subsample: 0.5. LR: 0.1. BP: 8.
Criteo-Full	**B:** ridge regression (rdg. reg.)	VW. L2 penalty: $1.5e^{-8}$. LR: 0.5. BP: 26.
MovieLens Regression	**B:** singular value decomposition (svd)	VW. Rank 10. L2 penalty: 0.001. LR: 0.015. BP: 18. Passes: 20. LR Decay: 0.97. PowerT: 0.
	linear regression (lin. reg.)	VW. LR: 0.5. BP: 22. Passes: 5. Early Terminate: 1.
	gradient boosted trees (gbt)	Sklearn. 100 trees with 8 leaves. Subsample: 0.5. LR: 0.1. BP: 8.
MovieLens Classification	**B:** singular value decomposition (svd)	VW. Rank 10. L2 penalty: 0.001. LR: 0.015. BP: 18. Passes: 20. LR decay: 0.97. PowerT: 0.
	logistic regression (log. reg.)	VW. LR: 0.5. BP: 22. Passes: 5. Early Terminate: 1.
	gradient boosted trees (gbt)	Sklearn. 100 trees with 8 leaves. Subsample: 0.5. LR: 0.1. BP: 8.
MSN.com	contextual bandit	VW. IPS context. bandit. LR: 0.02. BP: 18.

Table 3.2: Model parameters. The libraries and parameters used to train each model. The parameters not noted use library defaults. "LR" indicates the learning rate. "BP" indicates the hash featurization's bit precision (only applicable to raw models). "PowerT" exponent controls learning learning rate decay per step. "**B:**" indicates that the model will be used as a baseline. VW and Sklearn denote that the model was trained with Vowpal Wabbit [96] and scikit-learn [127], respectively.

tecting individual past observations with differential privacy adds 1% penalty to the accuracy, which remains within 5% of state-of-the-art models. (4) Pyramid's performance overheads are small.

Methodology

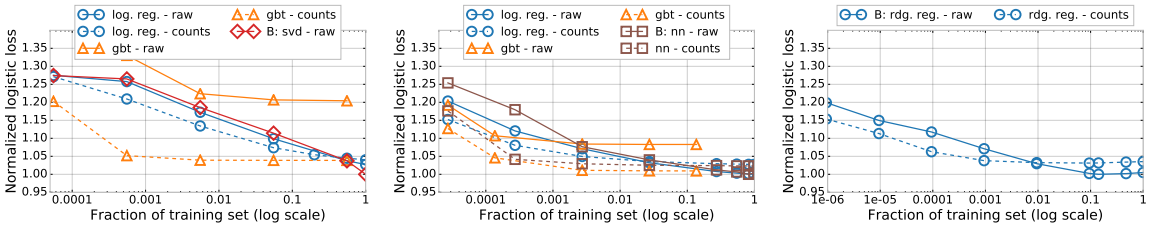
Workloads. Table 3.2 shows our apps, datasets, and baselines. We defer discussion of MSN to §3.5.

• *Criteo ad targeting.* Using two versions of the well-known Criteo ads dataset, we build a

binary click/no-click classifier. We use seven days of the Criteo ad click dataset amounting to 1.2 billion total observations. This dataset is very imbalanced with an approximate click rate of 3.34%. The second version of the Criteo dataset has 45 million observations, and was released as part of a Kaggle competition. In the Criteo Kaggle dataset, the click and non-click points were sampled at different rates to create a more balanced class split with a 25% click rate. Each observation has 39 features (13 numeric, 26 categorical), and 8 of the categorical features are high dimensional ($> 100K$ values). The numeric features were binned into 4 equal size bins for each dataset. As a baseline, we use a feed-forward neural network that performed well for the competition dataset[2], and we use ridge regression for the full dataset.

• *MovieLens movie recommendation.* Using the well-known MovieLens dataset, which consists of $22M$ ratings on $34K$ movies from $240K$ users, we build two predictors: (1) a regression model that predicts the user's rating as a continuous value in $[0, 5]$, (2) a binary classifier that predicts if a user will give a rating of 4 or more. As a baseline, we use the matrix factorization algorithm in Vowpal Wabbit (VW)[96]; algorithms in this class are state-of-the-art for recommender systems[95], although this specific implementation is not the most advanced.

Method. For each application, we try a variety of count models, including linear or logistic regression, neural networks, and boosted trees. We split each dataset by time into a training set (80%) and testing set (20%), except for the full Criteo dataset for which we use the first six days for training and the seventh for testing. On the training set, we compute the counts and train our models on windows of growing sizes, where all windows contain the most recent training data and grow backwards to include older data. This ensures that training occurs on the most recent data (closest to the testing set), and that count tables only include observations from the hot window or the past. We use the testing set to compare the performance of our count algorithms to their raw data counterparts and to the baseline algorithms. For all baselines, we apply any dimensionality reduction mechanisms (e.g., hash featurization[173]) that those models typically apply to strengthen them.



(a) **MovieLens classification** (b) **Criteo-Kaggle classification** (c) **Criteo-Full classification**

Figure 3.4: Normalized losses for raw and count algorithms. "B:" denotes the baseline model. Count algorithms converge faster than raw data algorithms, to results that are within 4% on MovieLens, and within 2% and 4% on Criteo Kaggle and full respectively.

Metrics. We use two model accuracy metrics.

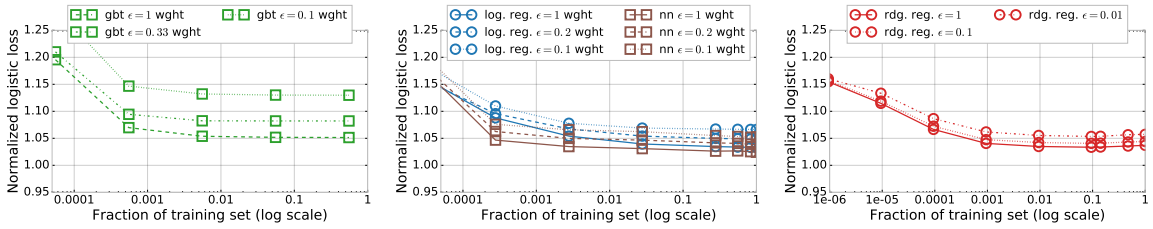
(1) The *average logistic loss* for classification problems with categorical labels (e.g. click/no-click). Algorithms predict a probability for each class and are penalized by the logarithm of the probability predicted for the true class: $-\log(p_{true\_class})$. Models are penalized less for incorrect, low-confidence predictions and more for incorrect, high-confidence predictions. Logistic loss is better suited than accuracy for classification problems with imbalanced classes because a model cannot perform well simply by returning the most common class.

(2) The *average squared loss* for regression problems with continuous labels. Algorithms make real-valued predictions that are penalized by the square of the difference with the label: $||prediction - label||^2$.

We conclude our evaluation with our experience with a production setting, in which we can directly estimate click-through rate, a more intuitive metric.

Result interpretation. All graphs report loss normalized by the baseline model trained on the *entire training data*. Lower values are better in all graphs: a value of 1 or less means that we beat the baseline's best performance; and a value > 1 means that we do worse than the baseline.

For completeness, we specify our baselines' performance: MovieLens classification matrix factorization has a logistic loss of 0.537; MovieLens regression matrix factorization has a squared loss of 0.697; Criteo-Kaggle neural network has a logistic loss of 0.467; and Criteo-Full ridge regression has a logistic loss of 0.136.



(a) **MovieLens boosted tree** (b) **Criteo-Kaggle algorithms** (c) **Criteo-Full ridge regression**

Figure 3.5: Impact of data protection. Results are normalized by the baselines. We fix $k = 1$ and vary ϵ, the privacy budget. Figure 3.5(a) and Figure 3.5(b) show results using the weighted noise (denoted wght). On MovieLens our weighting scheme is crucial to hide 1 observation. On Criteo we can easily hide 1 observation with little performance degradation and can hide up to 100 observations while remaining within 5% of the baseline.

Training Set Reduction (Q1)

Pyramid's design is predicated on count featurization's ability to substantially reduce training sets. While this method has long been known, we are unaware of scientific studies of its effectiveness for training set reduction. We hence perform a study here. The count models must converge faster than raw-data models (reach their best performance with less data), and perform on par with state-of-the-art baselines. Figure 3.4 shows the performance of several linear and nonlinear models, on raw and count-featurized data. We make two observations.

First, **training with counts requires less data**. On both Criteo and MovieLens the best count-featurized algorithm approaches the best raw-data algorithm by training on *1% of the data or less*. On Criteo-Kaggle (Figure 3.4(b)), the count-featurized neural network comes within 3% of the baseline when trained on 0.4% of the data and performs within 1.7% of the baseline with 28% of the training data. On Criteo-Full (Figure 3.4(c)), the count-featurized ridge regression model comes within 3.3% of the baseline with only 0.1% of the data, and within 2.5% when trained on 15% of the data. These results show that models trained on count-featurized data can perform close to raw models in both balanced and very imbalanced datasets (Criteo Full and Kaggle's respective click rates are 3% and 25%). On MovieLens (Figure 3.4(a)), the count-featurized boosted tree needs only 0.8% of the data to get within 4% of the baseline, or match the raw data logistic regression. Because counts summarize history and reduce dimensionality, they allow algorithms to perform well with very little

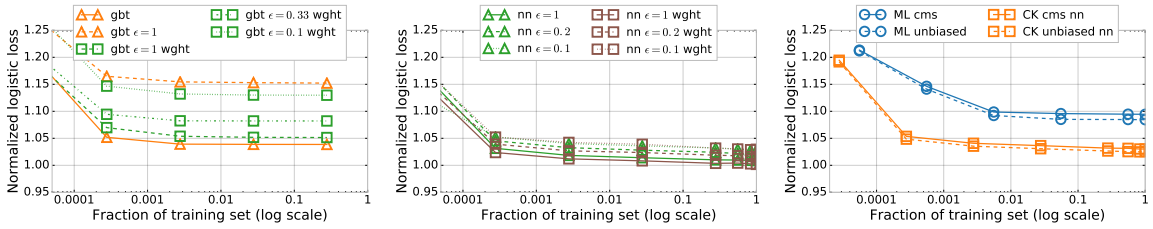
data. We say that they *converge faster* than raw data algorithms.

Second, **counts enable new models**. In Figure 3.4, the boosted tree performs poorly on raw data but very well on the count-featurized data. This reveals an interesting insight. The raw-data boosted tree uses a dimensionality reduction technique known as feature hashing[173], which hashes all categorical values to a limited-size space. This technique exhibits a trade-off: increasing the hash space reduces collisions at the cost of introducing more features, leading to overfitting. Count featurization does not have this problem: a categorical feature is mapped to a few new features (roughly one per label value). This lets us train boosted trees very effectively.

Past-Data Protection Evaluation (Q2)

We have shown that count-featurized algorithms converge faster than models trained on raw data. This allows Pyramid to keep, and thus expose, only a small amount of raw data to train ML models. However the count tables, while only aggregates of past data, can still leak information about past observations. To prevent such leaks, Pyramid adds differentially private noise to the tables. The amount of noise to add depends on the desired privacy guarantee, parameterized by ϵ (smaller is more private), but also on the number of features (see Table 3.2) and CMS hash functions (five here), through the formula from §3.3. In this section we evaluate the noise's impact on performance, as well as Pyramid's two mechanisms that increase data utility: automatic weighted noise infusion and the use of private count-median sketches. We also show the impact of the number of windows used, which defines the granularity at which past observations can be entirely dropped.

Impact of noise. Figure 3.5 shows the performance of different algorithms and datasets when protecting an observation, $k = 1$, with different privacy budgets ϵ (note the direct tradeoff between the two parameters: the noise is proportional to $\frac{k}{\epsilon}$). We find that Pyramid can protect observations with minimal performance loss. When $\epsilon = 1$, the boosted tree model on the



(a) **MovieLens boosted tree** (b) **Criteo-Kaggle neural network** (c) **Sketch comparison**

Figure 3.6: Impact of data protection (continued). Results are normalized to the baselines. We fix $k = 1$ and vary ϵ, the privacy budget. (a) Without the feature weighting trick the gradient boosted trees perform unacceptably poorly. (b) The weighting trick marginally improves the performance of Criteo-Kaggle models over equally distributing the privacy budget. (c) Private count-median sketch improves performance in both MovieLens (ML) and Criteo-Kaggle (CK) models with $\epsilon = 1$.

MovieLens dataset remains within 5% of the baseline with only 1% of the training data. The logistic regression and neural network models on the Criteo-Kaggle dataset perform within 2.7% and 1.8% of the baseline respectively, and the Criteo-Full ridge regression is within 3%. All Criteo models also come within 5% of their respective baseline with a privacy budget as small as $\epsilon = 0.2$.

The Criteo-Full ridge regression performance degrades less than models on other datasets when the noise increases. For instance, it degrades by less than 1% with ϵ going from 1 to 0.1, while the Criteo-Kaggle neural network loses 6.5%. This is explained by the fact that the amount of noise required to make a query differentially private is not related to the size of the dataset. The Criteo-Full dataset is much larger, so the additional noise is much smaller relative to the counts.

Weighted noise infusion. Weighted noise infusion is integral to the protection of past observations with minimal performance cost. Figure 3.6(a) shows the impact of noise on the boosted tree for the MovieLens dataset. Without weighting the privacy budget of different features, the model performs 15% worse than the baseline even for $\epsilon = 1$. With weighting, the MovieLens model performs at 5% of the baseline. The weighted noise infusion technique is thus critical to maintaining performance on the MovieLens dataset. Intuitively, this is because the users making the rating and the movie being rated are the most important features when predicting ratings. Most users rate relatively few movies, and a long tail of movies

Figure 3.7: Criteo-Full windows. The Criteo datasets can support 1K windows with reasonable penalty. Supporting more windows requires a scheme based on binary trees.

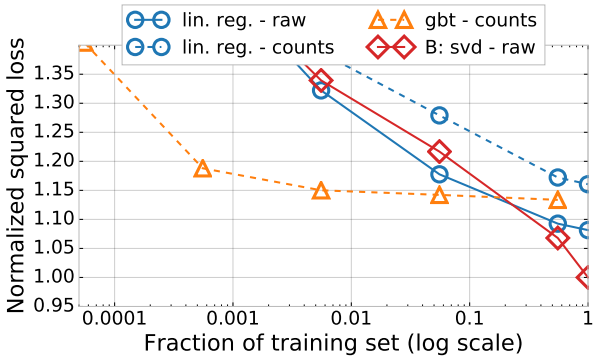


Figure 3.8: MovieLens regression. Linear regression algorithms are not amenable. Boosted tree converges quickly but does not match the baseline.

are rarely rated, so their respective counts are quickly overwhelmed by the noise when the privacy budget is equally distributed among all features.

The Criteo models do not depend as much on the weighting trick, since they do not rely on a few features with small counts. Noise weighting is still beneficial, though: e.g., the Criteo-Kaggle neural network gains about 0.5% of performance, as shown in Figure 3.6(b).

Private count-median sketch. Another technique that Pyramid uses to reduce the impact of noise is to switch to a private count-median sketch. As noted in §3.3, the count-min sketch will exhibit a strong downward bias when initialized with differentially private noise, because taking the minimum of multiple observations will select the most extreme negative noise values. The count-median sketch uses the median instead of the minimum and does not suffer from this effect. Figure 3.6(c) shows that when noise is added, the count-median sketch improves performance over the count-min sketch by around 0.5%, on MovieLens and Criteo-Kaggle.

When combined with weighted noise infusion, the private count-median sketch is less useful at first, since the noise is small on features with small counts. However, it provides an improvement for lower ϵ. For instance, the MovieLens boosted tree improves by 0.5% even after noise weighting for $\epsilon = 0.10$.

Number of windows. Another factor impacting accuracy is the number of count windows

kept to support granular retention policies. Figure 3.7 shows Criteo-Full's ridge regression for $k = 1$ and $\epsilon = 1$ while varying the number of windows. We observe that it is possible to support a large number of windows. On Criteo, we can support 1000 windows with little degradation, enough to support a daily granularity for a multi-year retention period. While we believe this granularity for retention policies should be enough in practice, we also simulated a binary tree scheme[31] that supports huge numbers of windows. We can see that on Criteo, this allows using $100K$ windows with a penalty similar to 10 windows using the basic scheme.

Count Selection Evaluation (Q3)

Without noise. We measure the performance of our algorithms when the featurization is augmented by MI-selected groups. We evaluate on MovieLens, as groups provided little additional benefit on Criteo. A total of 35 groups were selected by MI and given 10% of the privacy budget to share. When using these groups, the accuracy of the count boosted tree gets within 3% of the baseline with the same 0.8% of the data, 1% better than without feature groups. Logistic regression does not improve asymptotically but converges faster, getting within 5% of the baseline with 15% of the data instead of 22%. Thus, count selection selects relevant groups.

With noise. We also evaluate the impact of group selection on MovieLens with noise $k = 1$, $\epsilon = 1$. Logistic regression is not improved by the grouped features, but the boosted tree is still 1% closer to the baseline. Thus, the algorithm can still extract useful information from the groups despite the increased noise.

While these results are encouraging, we leave for future work the full investigation of how the improvement in accuracy gained from maintaining and using relevant groups is affected by the higher noise levels necessary to maintain a large number of count tables for fixed ϵ.

Performance Evaluation (Q4)

We evaluate Pyramid's overhead on Velox by measuring the median latency of a prediction request to Velox. We perform this evaluation using the 39-feature Criteo dataset. Figure 3.9 shows the median latencies and a breakdown of the time into four components: computing the prediction, unmarshalling the message into a usable form, performing count featurization, and other functions like the network and traversing the web stack. We show the results with and without count table caching in the application servers (§3.4). Without caching, prediction latency is around 200ms. Caching reduces it to 1.6ms, a 5% overhead with the total time dominated by the network and traversing the web framework used to implement Velox. Pushing count tables to the application servers is crucial for performance and does not significantly increase the attack surface.

Applicability Evaluation (Q5)

Pyramid works well for classification problems. We now consider another broad class of supervised learning problems: regression problems. In regression, the algorithm guesses a label on a continuous scale, and the goal is for the prediction to be as close to the true label as possible. Intuitively, count featurization should be less effective for regression problems, because it needs to bin the continuous label into discrete buckets.

Fig. 3.8 shows the performance of linear and boosted tree (nonlinear) regressions on the MovieLens dataset. We first observe that linear regression does worse on count-featurized data than on raw data. This is not surprising: count featurization gives the probability of each label conditioned on a feature. The algorithm cannot find a linear relationship between, say, $P(\textit{rating} = 3|\textit{user})$ and the rating. Indeed, the rating does not keep growing with this probability, it keeps getting closer to 3.

Nonlinear algorithms do not have this limitation. The boosted tree converges quickly and outperforms raw models trained on similar amounts of data until we reach 55% of the data.

Action	P. w/o cache	P. w/ cache	Velox
Featurization	99.22%	4.37%	N/A
Marshalling	0.04%	6.44%	7.06%
Prediction	0.01%	0.51%	0.63%
Network/Framework	0.73%	88.68%	92.31%
Total Latency	283.69 ms	1.65 ms	1.58 ms

Figure 3.9: Prediction Latency. Median time to serve a model prediction. Caching is crucial for Pyramid to achieve low overhead compared to Velox.

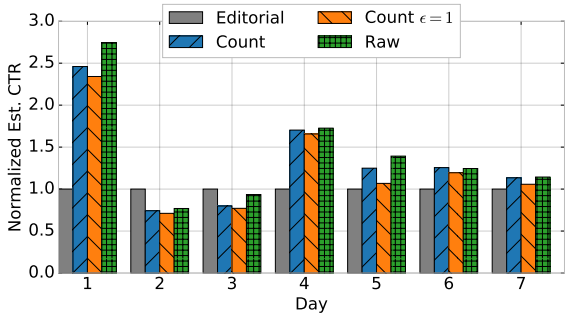


Figure 3.10: Estimated article CTR for MSN. The raw model, count model, and private count model are normalized against the estimated performance of human editors. The count models perform slightly worse than the raw models; all models outperform human editors on five out of seven days.

At that point, the boosted tree plateaus and never comes close to the baseline. Although we did not find good algorithms for this dataset, we suspect that some nonlinear algorithms may perform well on counts.

Count featurization is most reminiscent of the counts used by Naive Bayes classifiers[140], and there are workloads for which it is not suitable. For instance, count featurization requires a label and is thus not applicable to unsupervised learning. Other feature representations may be better suited to such types of models. Our choice of count featurization reflects its suitability to data protection in a practical system architecture.

Even in settings that are less amenable to Pyramid, such as online learning applications that avoid retraining, we found that Pyramid can perform well and help protect past observations, as we describe in the next section.

Experience with a Production Setting

In addition to public datasets, we also evaluated Pyramid on a production workload. One of the authors helped build MSN's news personalization service, which we used to evaluate three aspects: (1) How to adapt count featurization to a different type of learning, (2) how

Pyramid applies to this application, and (3) how Pyramid supports the application's workload evolution.

Adapting count featurization. MSN uses contextual bandit learning[97;12] (via the Decision Service[11]) to personalize the order of articles shown to each user so as to maximize clicks, based on 507 features of user demographics and past browsing history. This is a challenging scenario due to the large number of features and low click signal. Contextual bandit algorithms use randomization to explore different action choices, e.g., picking the top article at random. This produces a dataset that assigns a probability (importance weight) to each datapoint. The probabilities are used to optimize models offline and obtain unbiased estimates of their performance had they been run online[49;100;101].

Importance-weighted data have interesting implications for Pyramid. When updating the count tables with a given data point, Pyramid must increment the counts by $1/p$, rather than 1, to ensure they remain unbiased. This weighting also increases the noise required for differential privacy, because the sensitivity of a single observation can now be as high as $1/p_{min}$, where p_{min} is the minimum probability of any data point.

With these changes, we built a linear model on count-featurized data and compare it to the (linear) raw-data model used in production. Both models were trained using VW's online contextual bandit learner; in the production system, a snapshot of the model is deployed to application servers every five minutes.

Applicability. Our results suggest that in this application, selectivity is achieved naturally by retaining only the last day of data in the hot window and without the need for Pyramid's training set minimization. This is because news is highly non-stationary: new content appears every hour and breaking news influences people's short-term interests. As a result, even without Pyramid, training models on the last day of raw data is sufficient, and in fact better than training on more days. This is in contrast to the MovieLens and Criteo datasets, which are much more stationary and hence can benefit from Pyramid's training set reduction.

That said, even in non-stationary settings, Pyramid can still enhance data protection through its privacy-preserving counts. We compared the estimated click-through rate (CTR) of the count model (with and without noise) to the raw model across a seven-day period in April 2016. Figure 3.10 shows the results relative to the default article ranking by editors. Despite day-to-day variations, on average count models perform within 7% and 13.5% (with noise) of the raw model performance.

Support for workload evolution. We also assessed how Pyramid would support changes in MSN over time, without accessing the raw data store. MSN developers have spent hundreds (thousands) of human (compute) hours optimizing the production models. The changes include: tuning hyperparameters and learning rates, adding L1/L2 regularization, testing different exploration rates or model deployment intervals, and adding/interacting/removing features. For example, in some regions regulatory policies prevent certain user data from being collected, so they are removed and models are retrained. Pyramid supports all of the listed changes (§3.3) except adding new features/feature interactions.

3.6 Discussion and Analysis

We analyze Pyramid's security properties in the context of our threat model (§3.2), pointing out its limitations. A Pyramid deployment has three components: (1) A central repository of raw data in cold storage that is infrequently accessed and is assumed to be secure. Protecting this data store is outside of Pyramid's scope. (2) A compute/storage cluster used to train models, store the plaintext hot window, and to store and update count tables. (3) Numerous model servers storing trained models and cached versions of count tables.

We first examine the effects of compromising the cluster responsible for training models, maintaining the hot window, and storing the count tables. This will reveal the state of the count tables at time T_{attack}-Δ_{hot} by subtracting all observations residing in the hot window at T_{attack}. Property **P1** in §3.2 captures this exposure. However, the observations from the range

$[T_{attack}\text{-}\Delta_{retention}, T_{attack}\text{-}\Delta_{hot}]$ are protected through differential privacy (property **P2** in §3.2). We expect that the hot window (Δ_{hot}) will be small enough that only a small fraction of an organization's data will be exposed. Observations whose retention period ended before T_{attack} will have been erased, and the models will have been retrained to forget this information (property **P3** in §3.2).

In addition to the hot data, the adversary can siphon observations arriving in the interval $[T_{attack}, T^{end}_{attack}]$. Hence, the amount of data exposed depends on the time to discover and respond to an attack. The sliding nature of Pyramid's hot window gives the organization an advantage when investigating breaches. If an organization knows T_{attack} and T^{stop}_{attack}, it will be able to determine exactly which observations were exposed to the attacker and take the appropriate steps. Knowing these times is only required for post-attack auditing, not for protection of past data during the attack.

Under our current threat model, Pyramid does not protect data from multiple intrusions happening during the same time window. If an attacker accesses Pyramid's internal count tables, that attack is eradicated, and then gains access again at $T_{attack2}$ where $T_{attack2}$ follows T^{stop}_{attack}, the attacker will be able to compute the full fidelity count tables for updates that occurred during the time range $[T^{stop}_{attack}, min(T_{attack2}, T_{win\_end})]$ by subtracting the state of the count table at T_{attack} from the state of the same count table at $T_{attack2}$. T_{win_end} is the time when Pyramid finishes populating the count table it was populating at T^{stop}_{attack}. One approach to mitigate this attack is to require that Pyramid recomputes count tables after T^{stop}_{attack}, including reinitializing them with new draws from the Laplacian distribution. This will require an increased privacy budget but will still provide a privacy guarantee.

§3.5 demonstrates the need to cache count tables on the application model servers. Attackers that compromise an application server will gain access to the existing cached count table, trained models, and a stream of plaintext prediction requests (unlabeled observations). With access only to the application server the adversary will be able to calculate the dif-

ference between the existing count table and new count tables as they are replicated. The adversary will learn little because the difference between the cached count table and the newly replicated count table will be differentially private.

A key limitation of our system stems from our design choice to expose data for a period of time, while it is hot. Data is exposed through the hot data store, trained models, external predictions, and other states that may persist after the data is phased out into the differentially private count tables. There are three implications of this design choice. First, an adversary may monitor these states *before* actually mounting the full-system break-in that Pyramid is designed to protect against (so before T_{start}). §3.2 explicitly leaves this attack out of scope. Second, exposing the hot data in raw form to programmers and applications may produce data residues that persist after the data is phased out, potentially revealing past information when an attacker breaks in at T_{start}. For example, a programmer may create a local copy of the hot window at time T for experimentation purposes. While we cannot ensure that state created out-of-band is securely managed, the Pyramid design strives to eliminate any residues for state that Pyramid manages. This is why we enforce model retraining whenever the hot window is rolled over. And this is why we clarify in §3.3 that the count and weight selection mechanisms should incorporate differential privacy. Third, while the exposed hot data may be small (e.g., 1% of all the data), it may still reveal sufficient sensitive information to satisfy the attacker's goal. Despite these caveats, we believe that our design decision to expose a little hot data affords important practical benefits that would be difficult to achieve with a fully protected design. For example, unlike fully differentially private designs[110], our scheme allows training of *unchanged* ML algorithms with limited impact on their accuracy. Unlike encrypted databases[130;168], our scheme provides performance and scalability close to—or even better than—running on the raw, fully exposed data.

3.7 Related Work

Closest works. Closest to our work are the building blocks we leverage for Pyramid's selective data protection architecture: count featurization and differential privacy. *Count featurization* has been developed and adopted to improve performance and scalability of certain learning systems. We are the first to retrofit it to improve data protection, defining the protection guarantees that can be achieved and implementing them without sacrificing accuracy.

To implement these guarantees, we leverage *differential privacy theory*[51]. The typical threat model for differentially private systems[111;139;110] is different from ours: they protect user privacy in the results of a publicly released computation, whereas Pyramid aims to protect data inside the system, by minimizing access to historical data so its accesses can be controlled and monitored more tightly. For example, differential privacy frameworks (e.g., PINQ[111] and Airavat[139], adding privacy to LINQ and MapReduce respectively) ensure that the result of a query will be differentially private. However, these systems require full and permanent access to the data. The same holds for privacy-preserving recommender systems[110]. Pan-privacy[55;31;114] is a variant of differential privacy that holds even when an adversary can observe the system's internal state, a threat model close to ours.

Pyramid is the first to combine count featurization with differential privacy for protection.[1] This raises significant challenges at scale, including rampant noise with large numbers of count tables and damaging interference of differential privacy noise with count-min sketches. To address these challenges, our design includes two techniques: noise weighting and private count-median sketches. Prior art, such as iReduct[179] or GUPT[116], included a noise weighting scheme to allocate less of the privacy budget to queries with larger results. To our knowledge, we are the first to point out the limitations of CMS integration with differential privacy and propose private count-median sketches as a solution.

[1] Azure applies tiny levels of Laplacian noise to count featurization to avoid overfitting, but such low levels neither provide protection nor raise the challenges we encountered.

Alternative protection approaches. Many alternative protection models exist. First, many companies enforce a *data retention period*. However, because of the data's perceived benefit, most companies configure long periods. Google maintains data for 9-18 months[16]. Pyramid limits the data's exposure for as long as the company decides to retain it. Second, some companies *anonymize data*: Google erases the last byte of IP addresses in search logs after 6 months[65]. Anonymization provides very weak protection[120]. Pyramid leverages differential privacy to provide rigorous protection guarantees. Third, some companies enforce *access controls* on the data. Google's Sawmill strips out sensitive data before returning results to processes lacking certain permissions[23]. Given the push toward increased developer access to data[132;174], Pyramid provides additional benefit by protecting data on a needs basis.

Data minimization. Compact data representation is an important topic in big data systems, and many techniques exist for different scenarios. *Sketching techniques* compute compact representations of the data that support queries of summary statistics[41], large-scale regression analysis[105], privacy preserving aggregation[112]; *streaming/online algorithms*[119;150] process the data using bounded memory, retaining only the information relevant for the problem at hand; *dimensionality reduction techniques*[28] find a low-dimensional, faithful representation of the raw data, according to different measures of faithfulness; *hash featurization*[152] compacts high-cardinality categorical variables; *coresets*[64;13] are data subsets giving a good approximation for a given computation; *autoencoders* attempt to learn a compressed identity function[70].

We believe that this rich literature should be inspected for candidates for selective data protection. Not all mechanisms will be suitable. For example, according to our evaluation (Figure 3.4), hash featurization[152] does not yield sufficient training set reduction. And none of the mechanisms listed above appear to support workload evolution. The next section presents a few promising techniques we have identified.

Chapter 4

Pebbles: Data Management in Modern Operating Systems

4.1 Introduction

Despite recent high-profile failures in applications' management of our data[15], in the absence of system-level support for fine-grained data organization, we are forced to entrust them with our data. When users perform day-to-day data management activities – deleting individual emails, identifying specific data that was viewed, or sharing pictures – they are forced to rely on applications to behave properly. Yet, a 2010 study of 30 popular Android applications showed that 20 leaked sensitive data, such as contacts or locations[62]. Our own study of deletion practices within mobile apps, described later in this paper, revealed that 18 of 50 popular Android applications left information behind instead of deleting it. Notably, we found that until 2011, Android's default email application left behind the attachments of deleted emails while deleting the messages themselves.

Although a plethora of system-level data management tools exist – including encrypted file systems[81;74], deniable file systems[167], auditing file systems[67], or assured delete systems[129] – these tools operate at a single level of abstraction: *files*. Without a one-to-one mapping between user-relevant objects (for example, individual email messages in a mail client or documents in a word processor) and files, such systems provide poor granularity, preventing end-users from protecting individual objects that matter to them.

Consider Android's default email application: it stores each email's contents and to/from/-subject fields as several rows in a SQLite database (all emails are stored in the same DB, which is itself stored as a single file), attachments as files, and cached renderings of messages in different files. Such complex object-to-file mappings are typical in Android, as our large-scale measurement study of Android storage patterns shows (§4.3). Moreover, others have observed complex storage layouts in other OSes, such as OSX, where researchers have concluded that "a file is not a file" but a complex structure with complex access patterns[84].

Given the complexity of these object-to-file mappings, we ask: is it possible for system-level tools to support management and protection at the granularity of user-relevant objects?

Intuitively, this would require developers to specify the structure of their applications' persisted data to the operating system. Nevertheless, we observe that the high level storage abstractions included and predominant in today's operating systems – the SQLite relational database in Android and the CoreData object-relational mapper in iOS – bear sufficient structural information to recover these user-relevant data objects from unmodified applications.

We call these objects *logical data objects* (LDO), examples of which include an email (including its to, from, subject, body, attachments and any other related information); a mailbox including all emails in it; a bank account in a personal finance application; etc. We present *Pebbles*, a system that exposes LDOs to protection tools, without introducing any new programming models or interfaces, which can be prone to programmer error, slow adoption, or incompatibility with legacy applications.

We implemented Pebbles and several new protection tools based on it on the Android platform. Each of these tools provides protection at the LDO level, leveraging Pebbles to greatly simplify their development. Using Pebbles tools, users can mark objects from their existing applications to verify their proper deletion, protect their access from other applications, and back them up to the clouds they trust.

In a study of 50 popular Android applications, we found Pebbles to be highly effective in automatically identifying LDOs. Across these apps, object recognition recall was 91% and precision was 97%. In other words, in 91% of the cases, there was no leakage of data from user-visible objects to LDOs, and in 97% of the cases, there was no over-inclusion of extra data beyond user expectation in LDOs. Pebbles relies on several key assumptions based on common practices. Many of the cases in which Pebbles had poor accuracy, it could have been addressed had the developers followed these common practices.

Overall, this work makes the following contributions:

1. A study of over 470,000 Android apps, analyzing, for the first time at scale, the storage abstractions in common use today (§4.3). Our results suggest major differences

compared to traditional storage abstractions, which render file-level data management ineffective while creating untapped opportunities for object-level data management.

2. The first design and implementation of a persistent data object recognition system that requires no app changes (§4.3 and §4.4). Our design taps into the opportunities observed from our large-scale Android app study.
3. Four protection tools implemented atop Pebbles, demonstrating the power and value of application-level objects to protection tools (§4.3).
4. An evaluation of LDO construction accuracy with Pebbles over 50 popular applications from Google Play, showing it to be effective in practice (§4.5) and underscoring its well-defined failure modes (§4.6).

4.2 Motivation and Goals

We begin by presenting a set of example scenarios that highlight the need for fine-grained data management support within modern OSes.

Example Scenarios

Scenario 1: Object Deletion: Ann, an investigative journalist, has received an extremely sensitive email on her phone with an attachment that identifies her sources. To protect her sources, Ann does her due diligence by deleting the email immediately after reading its contents and restarting her phone to clean up any traces left in memory. Her phone is already configured with an assured-delete file system[129] that deletes data promptly upon request. Worried that the application might have created a copy of her data without her knowledge or control, she wonders: *Is there any remnant of that email left anywhere on the phone?* She is disappointed to realize that she has zero visibility into the data stored on her device. Weeks later, she learns that her fears were well-founded: the email app she is using contains a bug

that leaves attachments intact when an email is deleted.

Scenario 2: Object Access Auditing: Bob, a financial auditor, uses his phone for all interactions with client data while on field engagements. Recently, Bob's device was stolen. Fearing that his fingerprint unlock might not withstand motivated attackers[164], Bob asked his IT admin a natural question: *Has any of my clients' data been exposed?* The admin's answer was mixed. Although activity on Bob's phone was tracked by a remote auditing file system[67], the logs show that a file, `/data/data/com.android.email/cache/7dcee8`, was accessed immediately before the phone's wipe-out. The file stores the HTML rendering of an email, but no one knows *which* email. Bob is left wondering what he should disclose to clients about the potential exposure of their data, and to *which* clients, since neither he nor the IT staff can map that file to a specific client or email.

Scenario 3: Object Access Restriction: Carla, a local politician, uses her phone to take photos for professional purposes, but she has several personal photos on it as well. She uses a cloud-based photo editor to enhance her promotional photos before posting them. Due to the coarse-grained permissions model of her Android device, she must provide this photo editor with access to all of her photos in order to use it. Carla is concerned that the photo editor may be secretly collecting all the photos from her device, including several potentially sensitive photos that could be politically compromising.

Goals and Assumptions

The above hypothetical users, along with millions of real-life users of mobile technology, have a mental model of application-level objects that is not matched by current protection tools. Ann wants to ensure that a particularly sensitive email is deleted in full, including attachments, to, from, any related caches, and other fields; Bob wants to know the sender or contents of a compromised email instead of a meaningless file name; Carla wants to protect a few of her most sensitive photos from prying applications. Traditional protection tools,

such as file-based encryption, auditing, or secure deletion cannot fulfill these needs because the mapping between objects and files is application-specific and complex. The alternative, whole-disk encryption[160;1], does not provide the flexibility that these users need.

To support such object-level data management needs, we developed Pebbles, a system that automatically reconstructs application-level logical data objects (LDOs) from unmodified applications. Pebbles exposes these LDOs to any system-wide protection tool that could benefit from understanding application-level objects. An encryption system could use LDOs to support meaningful fine-grained protection as an extra layer on top of whole-disk encryption. An auditing system could use LDOs to provide meaningful information about an accessed component. An object manager could reveal to users which parts of an object are left after deletion. And a backup system could let users choose their most sensitive objects for backup onto a trusted, self-managed server, letting the rest be backed up into the cloud.

Goals. The Pebbles design was guided by three goals:

G1: Accurate and Precise Object Recognition: Pebbles objects (LDOs) must closely match application-level persisted objects. This includes: (a) *avoiding data leaks* (if an item belongs to an LDO it must be included), and (b) *avoiding data over-inclusions* (if an item does not belong to an LDO it should not be included).

G2: Meaningful Granularity: Pebbles must recognize LDOs that are meaningful to users, such as individual emails.

G3: No New Application APIs: Pebbles must not require app developers to use new APIs; it can recommend developers to follow existing common practices but must work well even if they do not precisely follow.

Our first goal is accurate and precise object recognition (*G1*). We aim to achieve (1) good object recognition *recall* by avoiding leaks and (2) good object recognition *precision* by avoiding over-inclusions. We acknowledge that perfect recall or precision cannot be guaranteed in either an unsupervised approach or in a supervised API approach with imperfect

developers, since a poorly written app could convolute data structure in a way that Pebbles cannot recover. However, we wish to formulate clearly all potential sources of leakage, to design mechanisms to address the leakages for most applications (§4.3), and to remind developers how they could avoid such leakages by following existing common practices (§4.6).

Related to G1, our second goal (*G2*) is to recognize relevant and meaningful LDOs. For example, in an email app, Pebbles should be able to recognize individual emails, not just coarse accounts with many emails. We note here that Pebbles identifies application-level objects that are persisted in stable storage, and we assume that those have a direct mapping onto the objects that users interact with and wish to protect.

G3 stems from our skepticism that developers will convert applications to use new security-related APIs or correctly use such APIs. However, we do expect that most developers will follow certain common practices (as evaluated in §4.3). Pebbles addresses this by leveraging application-level semantics already available within storage abstractions such as database schemas, XML structures, and the file system hierarchy. Pebbles also provides recommendations for developers which are rooted in already popular development practices (§4.6).

Threat Models and Assumptions. Pebbles is designed to support fine-grained data management – such as encryption, auditing, and deletion of individual emails, photos, or documents – within modern OSes. The specific threat model for a given protection tool depends on that tool's goal; however, Pebbles's mechanisms should bolster the guarantees applications can provide. In general, we assume that protection tools are *trusted* system-wide services. This is similar to assumptions made by encrypted file systems, assured-delete file systems, and other current fine-grained data management tools.

We also assume that mobile applications that create or have access to a particular object, or part thereof, will not obfuscate their data's structure or act maliciously against Pebbles. For example, they will not create their own data formats and will not willfully interfere with analysis mechanisms involved in object discovery. An application that has not yet been given

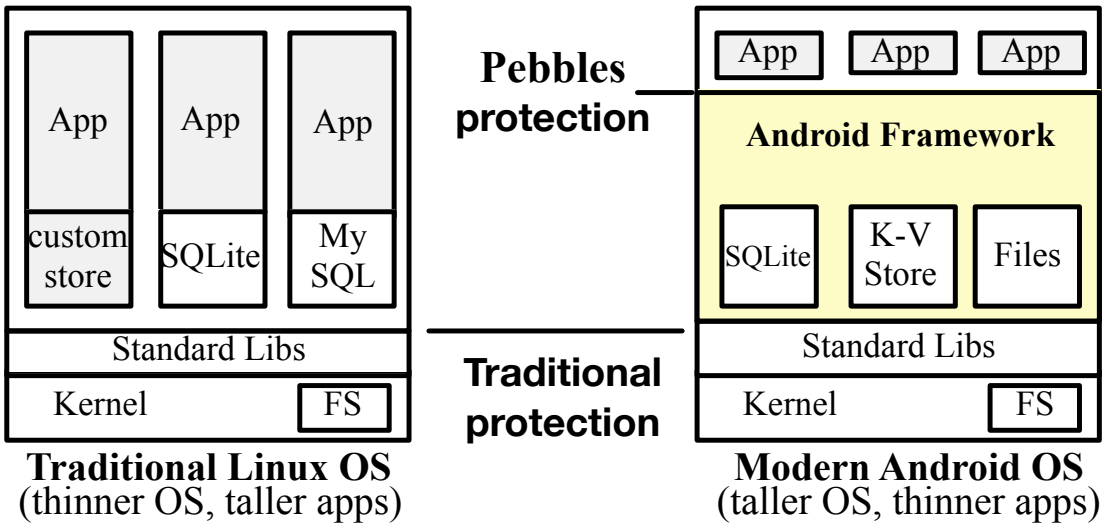


Figure 4.1: OS Storage Abstraction Evolution. Modern OSes provide higher-level abstractions for data management, yet protection is often at the traditional file level. Pebbles, our aligns data protection with modern abstractions.

access to data of a particular object, however, need not be trusted.

The scope of Pebbles is confined to those application-level objects that are persisted into a device's stable storage. We explicitly ignore attackers with access to either RAM or the underlying OS or hardware. If volatile memory protection is important, we recommend combining Pebbles with secure memory deallocation[40;39;GRSecurity], OS buffer cleaning[50], and idle in-RAM data eviction[162] mechanisms. We also assume that secure disk scrubbing[163;134] is deployed. In addition, while many modern applications include a cloud component, which stores or backs up data, Pebbles currently ignores that component. In the future, we plan to extend Pebbles LDOs to transcend the local and cloud environments.

While some may believe that users are incapable of dealing with fine-grained controls, we believe that there are many circumstances in which users want and are capable of handling *some* level of control, particularly for their most sensitive data. Evidence that users are capable of handling, and require, some level of control *when they feel it is important for them to do so* is available in prior studies[104;37]. Such evidence can also be gauged from the immense popularity of data hiding apps, such as Vault-Hide[NQ Mobile Security] and KeepSafe

Storage Abstraction	# Apps (of 98)	Example Apps
No storage	5	Cardio Trainer
DB only	43	**CWMoney**, Amazon, BestBuy, Browser, Calendar, Contacts, ColorNotes, EverNote
FS only	3	Exchange Rates
KV only	5	Google Talk, Biorythms
DB+FS	24	**OINote**, Angry Birds, DropBox, Gallery
DB+KV	1	Twitter
FS+KV	2	Adobe Reader, Temple Run
DB+FS+KV	15	**Email**, Antivirus, Amazon Kindle, Astro File Manager, Box, EBay

(a) Use of SQLite (DB), FS, and key/value (KV) store

Storage Library	# of Apps (of 476,375)
ORMLite	6,846 (1.4%)
SQLCipher	168 (0.3%)
DB4o	116 (0.2%)
H2	16 (0.0%)
Other 4 libs combined	38 (0.0%)

(b) Third-party library use

App	Object	DB/FS Use
Email (DB+FS+KV)	Email	to/from/date in one DB table; contents in another table; attachments in FS
	Mailbox	name/server/account in one DB table; includes emails; backup in kv
	Account	address/meta data in one DB table; includes mailboxes, emails
OINote (DB+FS)	Note	title/note/tags/ in one DB table; notes exported as files in /sdcard FS
CWMoney (DB only)	Expense	name/amount in one DB table
	Category	category name in one DB table; includes expenses
	Account	name/balance in one DB table; includes categories, expenses

(c) Example object structures

Figure 4.2: Storage API Usage in 98 Android Applications. (a) Number of apps that use the various storage abstractions in Android. Most apps use DB, but many also use FS and KV together with DB. (b) Use of eight other storage libraries among 476K free apps from Google Play. Third-party storage libraries are largely irrelevant. (c) Structure of sample objects in a few popular apps. Object structure is complex and spans multiple abstractions.

Vault[KeepSafe], which have garnered over 10 million downloads each and let users hide data, such as photos, contacts, and SMSes.

4.3 The Pebbles Architecture and Design

Pebbles aims to reconstruct application-level LDOs – emails and mailboxes in an email app, saved high scores in a game, etc. – from the bits and pieces stored across the various data storage abstractions without requiring application modifications.

Study: Android Storage Abstractions

The Pebbles design is motivated and informed by our high-level observation that storage abstractions within modern OSes are evolving in major yet unquantified ways. Figure 4.1 shows this evolution. Specifically, we hypothesize that the inclusion of high-level storage abstractions, such as the SQLite database in Android or the CoreData abstraction in iOS, has created a new "narrow waist" for storage abstractions that largely hides the traditional hierarchical file system abstraction. These new storage abstractions should bear sufficient

structure to let us reverse engineer application-level data objects from the OS's vantage point.

In this section, we perform a simple measurement study to gauge the use of these abstractions and extract useful insights to inform our design of Pebbles. We specifically ask the following questions:

Q1 What storage abstractions do Android apps use?

Q2 How do individual apps organize their data?

Q3 How are these abstractions used?

Background. Android provides three storage abstractions[Google] relevant to this paper:

1. *SQLite Database:* Stores structured data.
2. *XML-based Key/Value Store:* Stores primitive data in key/value pairs (also known as the SharedPreferences API).
3. *Files:* Stores unstructured data on the device's flash memory.

Methodology. We ran both *static* and *dynamic* experiments. Static experiments can be run at large scale but lack precision, while dynamic experiments provide precise answers but can only be run at small scale. For static experiments, we decompiled Android applications and searched their source code for imports of the storage abstractions' packages (e.g., `android.database.sqlite`). We ran large-scale, static experiments on 476,375 apps downloaded through a February 2013 crawl of Google Play[171], the main Android app market. For the dynamic experiments (over 98 apps), we installed Android apps on a Nexus S phone, manually interacted with them, and logged their accesses to the various APIs. These were some of the most popular apps, cutting across categories such as email clients, editors, banking, shopping, social, and gaming.

Results. *Q1 Answer: Apps primarily use SQLite, but use other abstractions as well.* Figure 4.2(a) classifies apps according to the Android-embedded storage abstractions they use during execution. It shows that the usage of Android-provided abstractions – SQLite (denoted DB) and the key/value store (denoted KV) – eclipses the traditional file abstractions

(denoted FS). Very few apps rely on the FS as their only storage abstraction (4/98). Almost half of the apps rely solely on SQLite for all of their storage needs (43/98), while almost all apps that have some local storage use SQLite (81/92). Even apps that one would consider to be primarily file-oriented (e.g., Astro File Manager, DropBox) use SQLite. A significant fraction of the apps (41/98) rely on more than one abstraction, and a notable fraction (15/98) rely on all three abstractions. This last result suggests a complex disk layout, a topic discussed further below. Overall, the most popular formations are: DB-only (43/98), DB+FS (23/98), and DB+FS+KV (15/98).

A related question is whether mobile apps use storage abstractions *other* than those provided by Android. Angry Birds, for example, stores game data and high scores in opaque binary files. We also searched the Internet for recommended Android storage options beyond those included in the OS, finding eight third-party libraries. We searched our 476K-app corpus for use of those libraries, and present the results in Figure 4.2(b). None of these libraries are popular: only 2% of the apps use even one of them. Our dynamic experiments found that none of these libraries are used and provided no indication of additional libraries that we might have overlooked.

Q2 Answer: Data objects span multiple storage abstractions. Figure 4.2(c) shows the structures of several logical data objects, representative of what users think and care about in various applications. It shows that objects often have complex structures that involve multiple storage abstractions. For example, Android's default email client, an example of the DB+FS+KV formation, stores various fields of the email object in two DB tables, attachments in the FS, and account recovery information in the KV. Object structure is fairly complex even for DB-only apps, such as CWMoney, a personal finance app, where a category includes metadata in one table and all expenses in another table. It thus spans multiple tables that are not linked together through explicit foreign keys. This suggests that protecting each storage abstraction separately will not work: any data protection abstraction at the end-user

object level must span multiple storage abstractions.

Q3 Answer: SQLite is the hub for data management. Given this complexity, a natural question concerns how one can even begin to build some meaningful protection abstraction. Using a modified TaintDroid (a popular data flow taint tracking system for Android[62]) version, we tracked the flow of data between storage abstractions, confirming that at least 70/81 apps that use the DB use it as a *central hub* for managing their data. By central hub, we mean that data flows mostly from the DB into the FS/KV (when they are used) or is accessed using pointers from the DB; an observation that was true for 27 of the 38 apps that use FS or KV in addition to the DB. For example, many apps, including Email, use files to store caches of rendered versions of data stored in SQLite (such as the body of an email) or blobs of data that are indexed and managed through SQLite (such as the contents of pictures, videos, or email attachments).

Thus, SQLite is not just frequently used; it is the central abstraction in Android that originates or indexes much of the data stored in the other abstractions. This result is encouraging because, intuitively, relational databases bear more explicit structure.

Implications for the Pebbles Design. Overall, our results suggest that while the storage abstraction landscape is fairly complex in Android, there is sufficient uniformity to warrant constructing of a broadly applicable object system. Such a system must detect relationships between objects stored in different abstractions. The results suggest that SQLite, a relational database that bears significant inherent structure, is the predominant storage abstraction in Android. Raw files, which lack such structure, are just used for overflow storage of bulk data, such as images, videos, and attachments. Based on these insights, we construct Pebbles, the first system to recognize application-level objects within modern operating systems without application modifications.

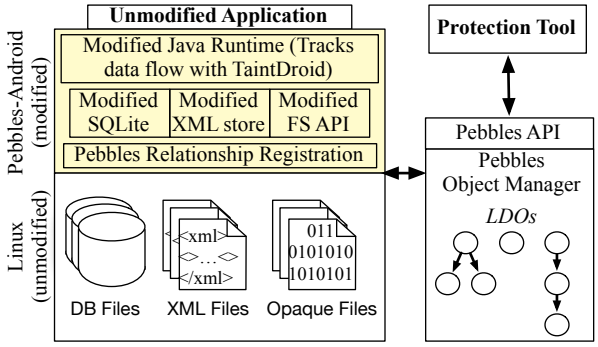


Figure 4.3: The Pebbles Architecture. Consists of a modified Android framework and a device-wide Pebbles Object Manager. The modified framework identifies relationships between persisted data items, such as rows, XML elements, or files. The Pebbles Object Manager uses those relationships to construct an object graph; nodes map to persisted data items and edges map to relationships.

Overview

Figure 4.3 shows the Pebbles architecture, which consists of two core components: (1) *Pebbles Android*, a modified Android framework that interposes on the various storage APIs, and (2) the *Pebbles Object Manager*, a separate device-wide entity for building object graphs and interacting with protection tools.

At the most basic level, the Pebbles Android framework understands units of storage (e.g., rows in DB, elements in XML, and files in FS) which become nodes in our object graph. The Pebbles Android framework then retrieves explicit relationships between these nodes and derives implicit relationships by tracking data flows between these units. The Pebbles Android framework registers these relationships with the Pebbles Object Manager using an internal registration API. The Pebbles Object Manager then stores these relationships, compiles a device-wide *object graph*, derives LDOs from the graph, and exports the LDOs to protection tools via the Pebbles API. LDOs are defined as follows: given a node in the graph (e.g., corresponding to a row in the `Email` table) an LDO is the transitive closure of the nodes connected to it. §4.5 evaluates Pebbles performance in terms of precision and recall. In the context of the graph, a failing of recall is missing nodes which should be included in a transitive closure ("leakage"); a failing of precision is including nodes which should *not* be included in a

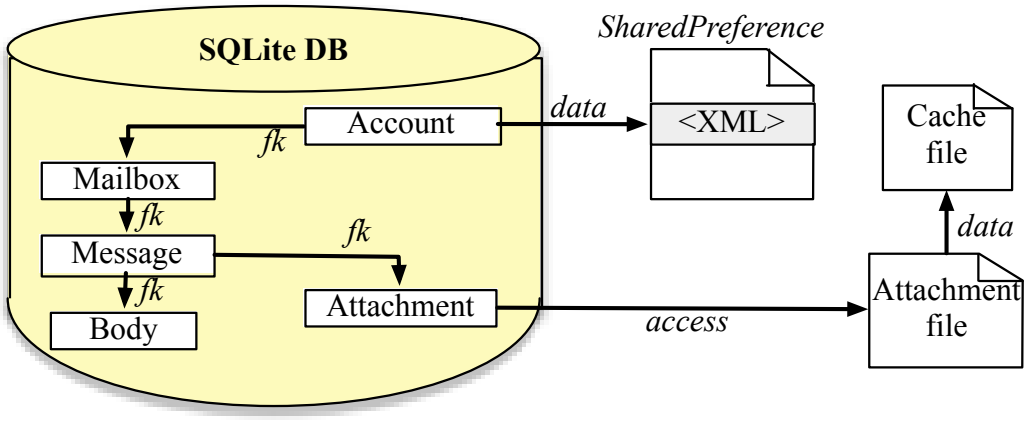


Figure 4.4: Android Email App Object Structure. A simplified object graph for one account with one mailbox, message, and attachment. Each node represents an individual file, row, or XML element, and each edge represents a relationship. While objects can be spread across the DB, FS, and Shared Preferences, the DB remains the hub for all data.

transitive closure ("over inclusion").

To provide a concrete example of the challenges faced by Pebbles, consider Figure 4.4, a simplified view of how data is stored by the default Android Email application. As described previously in §4.3, this app stores its data across all three storage abstractions: SQLite database, SharedPreference and individual files. Although a SharedPreference is used for account recovery, and several files are used to store an attachment and a cached rendering of it, the majority of the data is stored in SQLite.

Building the Object Graph

The object graph is the center of innovation in Pebbles: it directly represents Pebbles's understanding of the structure of an app's data and lets it construct LDOs. Each file, row, and XML element is assigned a 32 bit device-wide globally-unique ID (GUID) that is stored with the data item, which are hidden from and unmodifiable by applications. For database rows, the GUID is stored as an extra column in the row's table; for XML, it is stored as an attribute of each element; and for files, it is stored in an extended attribute. When a row, element, or file is read, the data coming from it is "tainted" with its GUID and tracked in memory using a modified version of the TaintDroid taint tracking system[62].

Pebbles builds the object graph incrementally by adding new files/rows/XML elements as nodes into the graph as they are created. It also adds directed edges (called *relationships*) between nodes in the graph as they are discovered. For example, when data tainted with one GUID is written into a file/row/XML element with another GUID, a relationship is registered. All nodes and edges of the graph are registered by the modified Android framework with the Pebbles Object Manager, where they are persisted in a database. We next describe the mechanisms used to build this graph, formalized in Figure 4.5.

Data flow propagation relationships: It is easy to see a strawman approach to detecting relationships between objects: when Pebbles detects that data tainted with node A's GUID is written into node B, it adds $A \leftrightarrow B$ to the object graph. This approach can capture all data flow relationships that occur within an application, regardless of the storage abstraction used. However, without precise information about the relationship between the two nodes, Pebbles is forced to assume the "worst case" scenario: that both nodes are part of the same LDO. Left unchecked, this so called taint explosion could eventually lead to all of an app's objects being included in the same LDO. Such behavior contradicts our primary goal of accurate and precise object recognition (G1). As we will see in §4.5, this naïve approach leads to unacceptably low precision (70%).

Utilizing explicit relationship information: Our next relationship detection mechanism relies on *explicit relationships* that directly communicate the programmer's view of his data structure to improve the precision. In a relational database, explicit relationships are defined in the form of foreign keys (FKs), which encode the precise relationship between two tables, based on primary keys (PKs). Interestingly, we can also extract a notion of foreign keys when relating DB rows to files: in some apps, the name of the file corresponds to the PK of the row to which it refers. Foreign keys encode the directionality of relationships, specifying for instance the difference between a "has-a" relationship and an "is-part-of" relationship. If node A has an FK to node B, then Pebbles adds the edge $A \rightarrow B$ (overriding any pre-existing bi-

directional edge detected from data flow propagation). In this way, foreign keys are precise but limited in coverage because they require programmers to specify them explicitly.

Increasing recall: Pebbles relies on one final relationship detection mechanism, *access relationships*. Access relationships can be seen as similar to data relationships, but while data relationships identify relationships as they are written to storage, access relationships identify relationships as they are read. Consider the case where an application has some data in memory that has not been synced to stable storage (and therefore is not yet tainted with any node's GUID). The app uses the data to generate the index for key-value object A and also writes that data into database row B. In the absence of explicit relationship information, we would hope that data propagation would detect the relation; however, it cannot because there is no data flow relationship when the data is written. We call this situation a *parallel write*, and resolve it by detecting data flow relationships when data is read in from storage: if data tainted with node A's GUID is used to access (read) node B, Pebbles adds $A \leftrightarrow B$ to the object graph. Again, this process is agnostic to the storage abstraction that the data is stored in, and relies only on data flow within the app. Access relationships can become an even greater source of imprecision than data relationships. For example, one could use data from one row, such as a timestamp, to select all the rows with that timestamp. Does that imply that all those rows should be considered as one object? Probably not.

Graph Generation Algorithm: Figure 4.5 defines the algorithm used to construct the object graph, based on the observation that the DB is the hub of all persisted data. Step (1) leverages data flow propagation to construct a base graph, while (2) refines that graph by applying explicit relationship information. Step (3) applies access based data flow propagation to increase recall, and (4) again refines that graph with explicit relationship information. §4.5 evaluates LDO construction accuracy and precision in detail.

Property 4.3.1. Apps define explicit relationships through FKs in DBs, XML hierarchies, or FS hierarchies

Property 4.3.2. The SQLite database is the hub of all persisted data storage and access

Object Graph Construction Algorithm:

1. Data propagation: If data from A is written to B, then $A \leftrightarrow B$
2. If possible, refine $A \leftrightarrow B$ to $A \rightarrow B$ using Prop 4.3.1
3. Access propagation: If data from A is used to read B, then $A \leftrightarrow B$
4. If possible, refine $A \leftrightarrow B$ to $A \rightarrow B$, again using Prop 4.3.1
5. Utilize Prop 4.3.2, eliminating access based data propagation relationships that do not include any DB nodes.

Figure 4.5: Object Graph Construction Rules.

LDO Construction and Semantics

After constructing the object graph using the above semantics, Pebbles extracts the LDOs. Within the graph, an LDO is defined as the set of reachable nodes starting with a given node (the root of the object). Consider the email graph (Figure 4.4), one can define a number of LDOs: an `Account` LDO, rooted in one `Account`-table row and containing multiple instances of five other row types, two files, and one XML entry; an `Email` LDO, rooted in one `Message`-table row and containing another row and one file, and so on. Although one LDO of each type is defined in the figure, in reality, there would be as many LDOs as there are instances of that type.

It is possible and correct for a single node to be part of multiple otherwise separate LDOs, in which case we say that the LDOs *overlap*. Consider, for instance, stateful accumulators (e.g. counts or sums over objects, stored in other objects), common resources (e.g. cache files that contain information about multiple objects), or log files.

Pebbles exposes LDOs to protection tools via the Pebbles API, which consists of two

Interface	Returned Objects
`getLDOContent(GUID, relevantOnly)`	LDO rooted at `GUID`
`getParentLDOs(GUID, relevantOnly)`	LDOs that contain `GUID`

Table 4.1: The Pebbles API for Accessing LDOs.

functions (Table 4.1). `getLDOContent` returns the LDO rooted at the given GUID and `getParentLDOs` returns the LDOs containing the given GUID. Protection tools may specify with each call if only LDOs that may be relevant to the end-user should be returned.

From User-Level Objects to LDOs

Both of these API methods require an "object of interest" as a parameter. Pebbles provides a framework for protection tools to allow users to directly select an object of interest (from the user interface), and then use that object for future API calls. In this approach, a user enables a "marking mode" from a device-wide menu item, and then touches the item that they are interested in. Through taint tracking, we can determine the internal GUID for the object that was selected, and return that GUID back to the protection tool. This feature makes designing user-centric protection tools very easy: the tool need not concern itself with determining which objects to protect.

The mechanisms described thus far are useful for building a graph of all of an application's objects, but does not yet include a way to identify those objects that are relevant to users. For instance, in our email application there is another table, "sync_state," that stores how recently an account was synchronized with the server. Sync_state should clearly not be considered its own LDO, as its existence is essentially hidden from the end-user – the user will likely consider whatever data is stored here as, logically, part of the account. Pebbles leverages its system-wide taint tracking to identify which nodes in the object graph are directly displayed on the screen, Pebbles marks those objects (and other LDOs of the same

type) as *relevant*. If an object is not relevant, then Pebbles will not allow it to be the root node of an LDO, instead including it as a member of the nearest parent node displayed on the screen.

Pebbles-based Tools

To showcase the value of Pebbles, we built four different applications that leverage its object graph.

Breadcrumbs: Auditing Object Deletion

Motivated by Scenario 1 in §4.2, Breadcrumbs lets users audit the deletion of their objects – such as emails or documents – by their applications. It uses Pebbles's primitives to track objects as they are being deleted and identify any breadcrumbs left behind by the application.

Users mark objects to audit for deletion (using Pebbles's object marking functionality), and then delete the object through their unmodified applications. They then open the Breadcrumbs application, which shows any persisted data related to recently tracked objects. In this way, users are not inundated with notifications about deletions and instead are only being presented with auditing information upon request. Figure 4.6 shows a screenshot of Breadcrumbs's output when the user deletes an email in the Android email application. It shows the attachment file left behind and provides meaningful information about the leakage. A brief predefined interval after the user deletes a tracked object, Breadcrumbs destroys all relevant auditing information to protect the confidentiality of the partially deleted object.

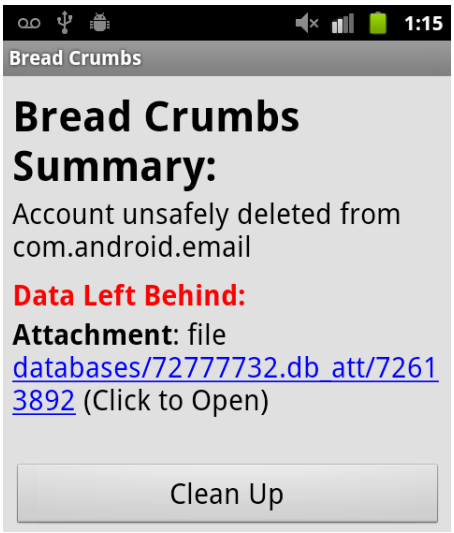


Figure 4.6: Breadcrumbs.

Algorithm 1 Breadcrumbs Pseudocode

```
function WASFULLYDELETED(LDO l) B →
    for all getLDOContent(l) as x do
        if x exists still then Add x → B
        end if
    end for
    for all B as x do
        Display x and getParentLDOs(x) to the user
    end for
end function
```

Algorithm 1 shows how Breadcrumbs uses Pebbles's APIs to obtain all information necessary to identify and provide meaningful information about data left behind. Given a selected UI object, Pebbles identifies the GUID of the LDO represented by that LDO (as described in the previous section), and then Breadcrumbs calls `getLDOContent` to get all of its parts. For any part that still exists in persistent storage – the attachment file in this case – it displays meaningful metadata about that node. For example, instead of just showing the file's path, which can be nondescript, Breadcrumbs uses Pebbles's `getParentLDOs` function to retrieve the parent node, presumably a row. It displays the row's table name ("*Attachment*" in Figure 4.6), providing more context for information left behind. While the specific user interface we chose for Breadcrumbs can be improved, this example underscores the great value protection tools like Breadcrumbs can draw from understanding application-level object structures.

Our evaluation of Breadcrumbs on 50 apps (§4.5), reveals that incomplete deletions are surprisingly common: 18/50 apps leave breadcrumbs or refuse to delete objects from the local device.

Breadcrumbs could also be a useful tool for developers. A developer could proactively use Breadcrumbs to ensure that they are responsibly handling their user's data.

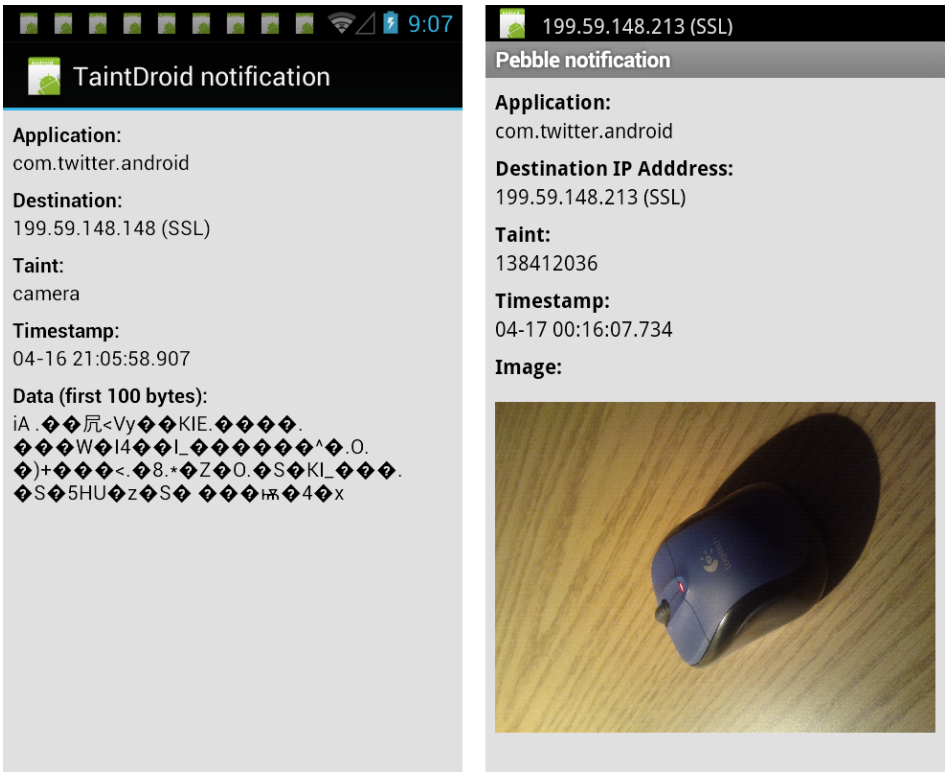


Figure 4.7: Alert Screenshots. (L): TaintDroid, (R): PebbleNotify.

PebbleNotify: Tracking Object Exfiltration

Inspired by TaintDroid's data exfiltration tool[62], we built PebbleNotify, a tool that tracks exfiltration at a more meaningful object level. TaintDroid reveals data exfiltration at a coarse granularity: it can only tell a user that *some* data from *some* provider was exfiltrated from the device, but not the specific data that was leaked. For instance, consider a cloud-based photo editing application. A user might expect this application to upload the photo being edited to a server for processing; however, he may be interested in checking that no other photos are exfiltrated. Shown in the left hand side of Figure 4.7, TaintDroid would warn the user that data related to *some* photo was uploaded, but not *which* photo or *how many* photos. PebbleNotify is a 500 line of code application built atop Pebbles that interposes on the same taint sinks as TaintDroid, but provides object-level warnings. §4.4 describes in somewhat greater detail the modifications that we made to TaintDroid to track individual objects with high precision. Shown in the right hand side of Figure 4.7, it leverages application-level data

structures exposed by Pebbles to give users meaningful, fine-grained information about their leaked objects.

PebbleDIFC: Object Level Access Control

As a logical extension to PebbleNotify, consider the case where rather than monitor the exfiltration of sensitive data, users want to prevent specific apps from having access to it. For example, in our previous example of a user using a cloud-based photo editing application, perhaps the user would rather simply prevent that photo editing app from having any access whatsoever to sensitive photos. PebbleDIFC supports this use-case by interposing on Android content providers, the mechanism used to share data between apps.

PebbleDIFC allows users to select individual objects that are sensitive, and then prevent them from being shared with other applications (in this case, photos). As with the rest of our protection tools, PebbleDIFC's implementation is straightforward. Before returning an object from a content provider, PebbleDIFC checks a table that maps apps to hidden objects, and prevents access to hidden objects.

HideIt: Object Level Hiding

Whereas PebbleDIFC allows objects to be permanently hidden from specific apps, *HideIt* supports a slightly different use case: allowing objects to be selectively hidden from all apps on the device, and then redisplayed at some later point, and perhaps hidden again later on. When objects are hidden (again, using Pebbles's marking mode), they are encrypted, and any record of their existence is filtered, by interposing on storage APIs. When objects are un-hidden, they are decrypted, and no longer filtered from API results. *HideIt* is intended for use-cases where small amounts of data need to be infrequently hidden from prying eyes, for instance, a parent lending their phone to their child.

Other Pebbles-based Tools

Although we designed and implemented Pebbles for Android, we believe that its object recognition mechanisms are applicable to other environments where a database is used as the hub of storage. In particular, we can imagine applying Pebbles as a software engineering tool to help developers understand either current or legacy applications where the database is the storage hub. A developer could use Pebbles to explore undocumented systems that do not make use of modern abstractions such as object relational mappers that would make the system easy to understand or to determine whether an application conforms to best practices and alert the developer if not. Understanding data structure from below the application could also enable testing tools and policy compliance auditing tools for cloud services[147]. We leave investigation of such applications for future work.

4.4 Implementation

We implemented Pebbles and each of the four above protection tools on Android 2.3.4 and TaintDroid 2.3.4. For Pebbles, we modify the SQLite, XML key/value store (a.k.a. SharedPreferences), and Java file system API to extract explicit structure, to intercept read-/write/delete operations, and to register relationships. We next review our TaintDroid changes, after which we describe some implementation-level details of object graph creation.

TaintDroid Changes. To support Pebbles, we made three modifications to TaintDroid: (1) we increase the number of supported taints from 32 to several million, (2) we implement multi-tainting to allow objects to have an arbitrary number of taints simultaneously, and (3) we implement fine-grained tainting. The first two TaintDroid changes are necessary to track every row, file, and XML element with a separate taint and are implemented with a technique recently proposed in the context of another taint tracking system[126]. We omit the details here for space reasons.

The third TaintDroid change is motivated by massive taint explosion that we observed due to TaintDroid's coarse-grained tracking. Specifically, TaintDroid stores a single taint tag per String and Array[62]. Deemed a performance benefit in the paper, this coarse-grained tracking is unusable in Pebbles: we observed extremely imprecise object recognition and application-wide LDOs due to this poor granularity. As one example, CWMoney, a personal finance application, has an internal array that holds selection arguments used in database queries. This causes all nodes selected by that query to be related, defeating any hopes of object precision.

To address this problem, we modify TaintDroid to add fine-grained tainting of individual Array and String elements. To implement fine-grained tainting we add a shadow buffer to the Dalvik ArrayObject that contains the taint of each element in the array. If implemented naively, the shadow arrays would likely double the memory required for each array. To minimize the memory overhead from the shadow arrays we allocate the shadow array only when a tainted element is inserted into the array. This same optimization is implemented in[42]. Intuitively, only a small fraction of arrays in an device's memory should contain tainted elements (3-5% according to our evaluation). §4.5 shows that this lazy shadow array allocation significantly reduces the memory overhead of precise fine-grained tainting. We release our changes open source as a patch for TaintDroid.

Object Graph Implementation. The Pebbles graph is populated incrementally during application execution and persisted in a central database on the data partition so the graph does not need to be regenerated on each reboot. Applications interact with the Pebbles API through the Pebbles Object Manager that runs as part of the central system server process. Graph edges are generated on read and write operations to SQLite, shared preferences, and the file system. On read and write operations that generate new edges, requests for edge registration are placed on a queue within the application's memory space. This lets Pebbles perform bulk asynchronous registrations off of the main application thread improving application in-

teractivity even during periods of heavy edge creation. In its current implementation the registration queue is not persisted to stable storage so it will be lost on application crashes or restarts. This is a potential attack vector that does not fall under the threat model for non-malicious applications.

4.5 Evaluation

We evaluate Pebbles over 50 popular applications downloaded from Google's Android market on a Nexus S running our modified version of Android 2.3.4. We seek answers three key questions:

Q1 *How accurate and precise is object identification in Pebbles?*

Q2 *What performance overhead does it introduce?*

Q3 *How useful are Pebbles and the tools running atop?*

Application Workloads. We chose 50 test applications from the top free apps within 10 different Google Play Store categories, including Books and Reference, Finance, and Productivity. We looked at the top 30 most popular applications within each category (by number of installs) and selected those that used stable storage. We also added a few open-source applications (e.g., OINote). The resulting list included: Email (Android's default email app), OINote (open-source note app), Browser (Android's default), CWMoney (personal finance app), Bloomberg (stocks app), and PodcastAddict (podcast app). For each application, our workload involved exercising it in natural ways according to manual scripts. For example, in Wunderlist, a todo list app, we created multiple lists, added items to each list, and browsed through its functions.

Pebbles Precision and Recall (Q1)

We measure the precision and recall of our object recognition by identifying how closely LDOs match real, application-level objects as users perceive them. We manually identified 68 potentially interesting LDO types across 50 popular applications (e.g., individual emails, folders, and accounts in the default email app; individual expenses, expense categories, and accounts in the CWMoney financial app). We evaluated whether Pebbles correctly identifies those objects (no leakage or over-inclusions). Recall measures the percentage of LDOs recognized without leakage; precision measures the percentage of LDOs recognized without over-inclusion.

To establish ground truth about LDO structure, we first populated the application with data and took a snapshot of the phone's disk, S_1, prior to creating the target object. Then, we created the object and took a second snapshot of the disk, S_2. The ground truth is the diff between S_2 and S_1 after manually excluding differences that are unrelated to the objects (e.g., timestamps in log files that differ between the two executions). We then exercised the application as thoroughly as possible so as to capture any edges that Pebbles might detect. To measure accuracy, we compare Pebbles-recognized LDOs to the ground truth; if identical, we declare accurate recognition for that application and object.

Table 4.2 shows whether Pebbles correctly and precisely detects these LDOs. For comparison, we also evaluated the precision and recall of a basic approach, which represents perhaps the current state of the art: detecting relationships between files using just taint tracking and not using additional file structure to refine the granularity of objects. Pebbles correctly identifies 60 of the 68 objects across these 50 apps, without requiring any program modifications. Of the eight incorrectly identified objects, six were not correctly detected and two were not precise.

In each case that Pebbles failed to properly detect all components of the object (i.e., where it failed in recall), the leakage was due to a non-standard database specification. For instance,

		Pebbles		File Tainting Only	
Application	**LDO**	**Detected**	**Precise**	**Detected**	**Precise**
Email	Account	Y	Y	Y	**N**
	Mailbox	Y	Y	Y	**N**
	Email	Y	Y	Y	**N**
OINote	Note	Y	Y	Y	**N**
Browser	History Item	Y	Y	Y	**N**
	Bookmark	Y	Y	Y	**N**
CWMoney	Account	Y	Y	Y	**N**
	Category	Y	Y	Y	**N**
	Expense	Y	Y	Y	**N**
Bloomberg	Stock	**N**	Y	Y	**N**
	Chart	Y	Y	Y	**N**
Podcast	Podcast	Y	Y	Y	**N**
	Episode	**N**	Y	Y	**N**
50 Total	68 Total	62/68 (91%)	66/68 (97%)	68/68 (100%)	0/68 (0%)

Table 4.2: LDO Precision and Recall. Sample applications and objects tested for object recognition precision and recall. "Y" indicates that an LDO was identified without leakage (column "Detected") or without over inclusion (column "Precise"). If an LDO has "Y" in both columns, its recognition is deemed correct. As expected, Pebbles performs far better than a straw man approach of treating entire files as a single LDO.

in the case of the app "ColorfulBudget", users can group expenses into categories, but Pebbles did not always properly detect the relationship between an expense and its category. Best practices would dictate that in such a case, all categories would be listed in a single table with a primary key (PK), and then each expense would contain a foreign key (FK) to reference the category's PK[27]. Traditionally this PK is an integer, to significantly increase lookup speed and decrease the amount of space needed to store any references to it[27]. However, in its current implementation, this app uses the actual name of the category as a key into the category table, without declaring such a dependency. Therefore, if a new category is created simultaneously with the creation of a new expense, we will experience a parallel write: there will be no data dependence when the category is inserted and when the expense is inserted, since the category did not yet exist in storage. Moreover, since the relationship is not declared in the app schema as an FK, explicit relationship mechanism will not detect it.

While our access-based technique will largely eliminate this problem, there is still a gap when data is written but never read back. In these scenarios, such relationships could never be detected. Had these apps explicitly declared their DB relationships (e.g., in the above case

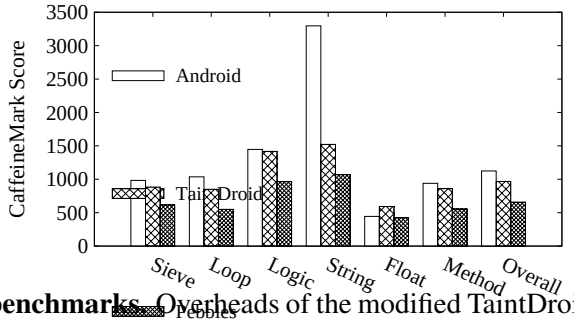


Figure 4.8: Java Microbenchmarks. Overheads of the modified TaintDroid on the Java runtime with CaffeineMark, a standard Java benchmark. Higher values are better. Overheads on top of TaintDroid are 28-35%.

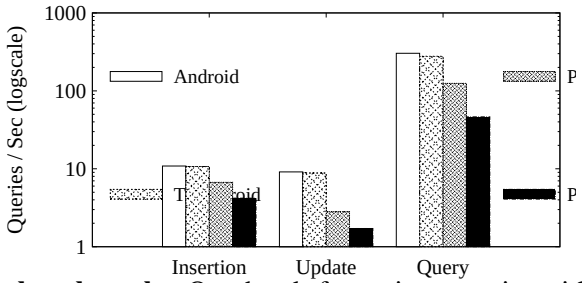


Figure 4.9: SQLite Microbenchmarks. Overheads for various queries without and with relationship registrations.

by referencing each category by its PK), Pebbles would accurately recognized the objects.

As an example of Pebbles failing in precision (i.e., including additional objects as part of an LDO), consider the "Evernote" note taking app. Each time a notebook is updated, text in a SharedPreferences node is updated to reflect the newest notebook, creating a data dependency between the SharedPreference and the notebook. In this way, each notebook can become related to each other because Pebbles currently does not break data dependencies when text is updated. The only way that relations are broken in Pebbles is if an explicit relationship exists and is removed.

Without requiring any modifications to applications, Pebbles is able to achieve up to 91% recall or 97% precision. The straw man approach of utilizing only taint tracking (without knowledge of file structure) showed perfect recall (100%), and a complete failure in precision (0%). In other words, there were *no cases* of a single logical object stored in a single file. Overall, our results confirm that an unsupervised approach to application-level object recognition from within the OS works well, especially if schemas are relatively well-defined.

Performance Evaluation (Q2)

To evaluate Pebbles performance overheads, we ran two types of benchmarks: (1) *microbenchmarks*, which let us stress various components of our system, such as the computation and SQLite plugins; and (2) *macrobenchmarks*, which let us quantify our system's performance impact on user-visible application latency. Pebbles is built atop the taint tracking system TaintDroid[62], with several modifications made to increase taint precision (as discussed in §4.4). Therefore, we evaluate the performance overhead of Pebbles in comparison to both TaintDroid and to a stock Android device.

Microbenchmarks. Our first experiments evaluate the overhead of Pebbles with the Java benchmark CaffeineMark 3.0[Pendragon Software Corporation] and are shown in Figure 4.8. We ran the six computational benchmarks and find that Pebbles decreases the score by 32% compared to TaintDroid, which itself decreases the score by 16% compared to Android. The majority of this overhead comes from modifications to support more than 32 taints in Pebbles: TaintDroid combines tags by bitwise OR'ing, but Pebbles supports 2^{32} distinct taint markings, which are maintained in a lookup table. Pebbles also stores taint tags per individual array element, whereas TaintDroid stores only one taint tag per array, creating an additional overhead for Pebbles array-heavy benchmarks.

Pebbles also incorporates modifications to SQLite to detect and register relationships between rows with the Pebbles service. To evaluate the overhead, we compared the latency of simple, constant-size `SELECT`, `INSERT`, and `UPDATE` queries on an Pebbles-enabled Android versus Android. Figure 4.9 shows query overheads when the query involves a relationship registration (59-168%) and when it does not (158-553%). No-registration queries – the cheapest to Pebbles – will likely be the common case for read-mostly workloads. For example, a document may be read many times, but relationship registration occurs only once. Moreover, batching and asynchronous-registration optimizations will likely help alleviate the overheads. The XML-based key/value store exhibits similar behavior, although we suppress

App	Activity	Base	TDroid	Pebbles	Overhead
Email	Launch	196.8	202.1	260.0	63.2 ±1.11
	Load Email	211.6	253.6	463.6	252.0 ±1.64
OINote	Launch	182.6	229.4	219.7	37.2 ±1.58
	Load Note	59.5	70.2	84.9	25.4 ±0.14
Browser	Launch	96.5	124.0	148.1	51.6 ±1.63
	Load (iana)	154.0	209.3	395.3	241.4 ±2.26
	Load (CNN)	778.9	862.7	1443.1	664.2 ±17.56
	Load (GNews)	951.3	1023.5	1311.2	359.9 ±10.75

Table 4.3: Application Performance. Operation runtimes and overheads in milliseconds. 95% confidence interval shown for overhead. Base is the Android baseline, TDroid is TaintDroid.

concrete results.

Application-Level Performance. The above workloads are micro-benchmarks that stress the various components but do not necessarily relate to user-perceived performance impacts. To measure the impact of Pebbles on user-perceived interactivity, we evaluated the runtimes for various operations with three popular applications: Email, Browser and OINote. For Email, we look at app launch times and email reads; for Browser, we load the simple IANA homepage and the rich CNN and Google News pages over a local network; and for OINote we read a note. All network access occurred over USB tethering to a host running a caching proxy; timing information excludes cache warmup. Table 4.3 shows the results in milliseconds. In almost all of the cases, overhead was less than 250ms. We saw more overhead and variation when rendering multimedia heavy web pages.

Memory Overheads. The modifications to TaintDroid to add fine grained tainting adds a memory overhead to the running system. We measure system wide memory usage while exercising three applications (Email, OINote, and Browser) with a similar workload as above. Without lazy memory allocation of array taint vectors (see §4.4), Pebbles's system-wide memory overheads are high: 188MB, 70MB, and 119MB, respectively, compared to TaintDroid. With lazy memory allocation, Pebbles exhibits much lower system-wide overheads: 34MB, 16MB, and 29MB, respectively. Although still higher than TaintDroid's own overhead of around 7MB for these applications, we believe Pebbles overheads are acceptable given devices' increased memory trends.

Application	Object Deletion Leakage
Email	Attachments remain after email/account deletion
ExpenseManager	Expenses remain after associated category deleted
Evernote	Notes/notebooks remain in database after deletion
On Track	Measurements remain after deleting category
14 other apps	21 LDO types unsafely deleted

Table 4.4: Breadcrumbs Findings. Shows samples of unsafe deletion in various applications.

Case Study Evaluation (Q3)

Breadcrumbs. Using our Breadcrumbs prototype we evaluated deletion practices of 68 types of LDOs across 50 applications. Of the 50 applications, 18 of them exhibited some type of deletion malpractice.

Table 4.4 shows sample deletion malpractice. There were several cases where data from one LDO was written into another another and not cleaned up later. There were also several applications that did not delete items at the users' request, instead simply removing them from the user interface. We observed this in applications that heavily rely on cloud storage such as Wunderlist, a popular cloud-backed todo list application.

PebbleNotify. To evaluate PebbleNotify, we compared its output to that of TaintDroid Notify. When TaintDroid Notify detects that data tainted with a value from one of the selected sources is exfiltrated, it notifies the user with the application that is responsible for the network connection, the destination, the data source, the timestamp, and the first 100 bytes of the packet. This is useful metadata but it won't help a user learn specific information about the data being exfiltrated such as which picture or specific contact is leaving the device. We found that PebbleNotify was more informative because it shows a summary of the data being exfiltrated, and not just the metadata presented by TaintDroid Notify. PebbleNotify was particularly useful in the case of image exfiltration because it displays a thumbnail of the image being sent.

PebbleDIFC. We integrated PebbleDIFC with the Android Media Provider and evaluated it by using it to mark several photographs on our device as sensitive (i.e., to prevent them

from being shared). We then verified that those photos were not visible to applications other than the default Gallery application. We found that for this use case, PebbleDIFC has perfect accuracy: every photo that was marked was hidden, and no additional photos were hidden.

HideIt. We evaluated *HideIt* against many applications and largely found it to be effective. In our evaluation, we interacted with the application, populated it with data, and then marked a subset of the data as private so the application no longer had access. Interestingly, in most cases apps behaved as hoped when individual data objects were hidden and then again returned. There were however several cases where apps crashed when they expected some data to still exist, but was removed. We are interested in performing further investigations of the applicability of *HideIt*.

Anecdotal User Experience

To gain experience with Pebbles, the primary author carried it on his Nexus S phone for about a week. He primarily used the Email, Browser, Gallery, Camera, and PodcastAddict apps. We report two anecdotal observations from this experience. First, applications exhibit noticeable overhead during periods of intense I/O, such as on initial launch or when applications populate or refresh local stores. During regular operation we observed overheads that are anecdotally similar to ones exhibited by running Android 4.1 (a 2012 OS) on our Nexus S (a 2010 device). Second, to check if object recognition remains accurate over time, we examined at the end of the week the structures of a sample of the objects in our applications (e.g., emails, folders, photos, browser histories, and podcasts). We saw no evidence that object recognition degraded over time due to taint explosions or other potential sources of imprecision for Pebbles. Objects grew naturally; email folders grew in size to include relevant new email objects and they remained accurate.

Summary

Overall, our results show that: Pebbles is quite accurate in constructing LDOs in an unsupervised manner (*Q1*), performance remains reasonable when doing so (*Q2*), and data management tools can benefit from Pebbles to provide useful, consumer-grade functions to the users (*Q3*). In our experience, Pebbles either consistently identifies objects of a particular type (e.g., all emails, all documents, etc.), or it does not. Whether it works depends largely upon the application's own adherence to some common practices (described in the next section). When Pebbles works for all object types of an application, Pebbles can provide the desired guarantees under our threat model. And even when Pebbles is incomplete, it can still support transparency applications, improving visibility into data (mis)management of applications. Our accuracy results show that Pebbles discovers all object types in 42 out of 50 applications correctly (no over-inclusions/leakages). We leave development of tools to identify whether an application matches the Pebbles assumptions for future work.

4.6 Discussion and Analysis

Pebbles leverages the structure inherently present in the storage abstractions commonly used on Android to identify LDOs. More formally, Pebbles assumes the usage of the following best practices:

R1: *Declare database schemas in full:* Given that the database is becoming the central point of all storage in modern OSes, having a well-defined database schema is important and natural. 42/50 apps we have evaluated in §4.5 meet such requirements sufficiently for Pebbles to work perfectly for them.

R2: *Use the database to index data within other storage systems:* A common programming pattern is to create a parent object (e.g., a message) in the database, obtain an auto-generated primary key, and then write any children objects (such as message body,

attachment files) using the PK as a link. 47/50 apps use this pattern. We strongly recommend it to any programmers who need to store data outside the DB.

R3: *Use standard storage libraries or implement Pebbles storage API:* To avoid precision lapses, we recommend that apps use standard storage abstractions. As §4.3 shows, most apps already adhere to this practice: most apps use exclusively OS-embedded abstractions.

Relative to our evaluation of 50 apps, 39/50 adhere with all three recommendations, and 50/50 adhere with at least one of them. Pebbles' performance could suffer for apps that do not follow any of these recommendations. However, we believe that each recommendation is sufficiently intuitive and rooted in best practices to not impose undue burden.

4.7 Related Work

Taint Tracking for Protection and Auditing. Taint tracking systems (such as[39;82;138;181;121;17;187]) implement a dynamic data flow analysis that has been applied to many different context such as privacy auditing[39;62;186], malware analysis[121], and more[187;17]. TaintDroid[62] provides taint tracking of unmodified Android applications through a modified Dalvik VM, a system that Pebbles builds upon for its object graph construction. To our knowledge, Pebbles is the first system to use taint tracking to discover data semantics of objects and provide a higher level abstraction with which to reason about and enforce such security properties.

Several systems utilize taint tracking to provide fine grained data protection and auditing. In each of these cases, however, a burden lies on the application developers to add hooks to identify relevant data structures to protection tool developers – a burden that could be lifted by Pebbles. For instance, CleanOS aims to minimize data exposure on a mobile device by automatically encrypting its "sensitive data objects" (SDOs) when not under active use[162]. The LDO abstraction is perhaps to some extent inspired by the SDO; however, SDOs must

be manually specified by application developers, whereas LDOs are automatically identified and registered by Pebbles. Pebbles could be used to automatically identify SDOs, without requiring developer interaction.

Distributed information flow control (DIFC) systems such as Laminar[138], Asbestos[170], and Resin[181] let developers associate data with labels, and then allow either developers or end-users to specify security policies that apply to different labels. Taint tracking is performed during application execution to ensure that labels are propagated to derived data. Pebbles could be used to eliminate the need to statically annotate data with labels in code, instead automatically applying labels to LDOs as users request them. PebbleDIFC demonstrates the feasibility and power of such a system.

Related to taint tracking, data provenance[118;117;Seltzer] is close in spirit to logical data objects. It tracks the lineage of data (e.g., the user or process that created it). It has been proposed to identify the original authors of online information, to facilitate reproduction of scientific experiments[Seltzer], detect and avoid faulty data propagation in clouds[118], and others. It has to our knowledge never been used as an OS protection abstraction.

Fine-Grained Protection in Operating Systems. Many systems have been proposed in the past to support fine-grained, flexible protection in operating systems. Some of the earliest OSes, such as Hydra[178] and Multics[141], provided immense protection flexibility to applications and users. Over time, OSes removed more and more flexibility, being considered too difficult for programmers. Our goal is to eliminate the programmer from the loop by having the OS identifying objects.

More recently, OS security extension systems, such as SELinux[SELinux] and its Android version, SEAndroid[SEAndroid], extend Linux's access control with flexible policies that determine which users and processes can access which resources, such as files, network interfaces, etc. Our work is complementary to these, being concerned with external attacks, such as thieves, shoulder surfing, or spying by a user with whom the device has been willfully

shared. Our abstractions, might, however, apply to SEAndroid to replace its antiquated file abstraction.

Securing and Hiding Data. Many encryption systems exist, operating largely at one of two levels of abstraction: block level[113;167;1] and file level[81;74]. A drawback to such encrypted file systems is that it forces users to consider data as individual files, while logically there may be multiple objects that the user is interested in in a single file. Pebbles allows protection tool developers to provide a far finer level of control (at the object level) than these existing systems (at the file level).

Some protection tools are already operating at a higher level of data abstraction. These applications, such as Vault-Hide[NQ Mobile Security] and KeepSafe Vault[KeepSafe], allow users to hide specific types of data, including photos, contacts, and SMSes. However, they only plug into a handful of supported apps and cannot provide generic protection for all apps. Pebbles aims to effect a similar level of control, but without requiring specialized work by protection tool developers to support specific applications.

Inferring Structure in Semistructured Data. Discovering data relationships is a key aspect of our work. Other have worked on inferring data relationships in various context: foreign key relationships in databases to improve querying[137;185] and file relationships in OSes to enhance file search[157]. However, Pebbles can also infer relations among files, as well as other higher-level storage abstractions within modern operating systems. To perform such broad relationship detection, Pebbles differs significantly from other relationship detection systems in that it also leverages taint tracking.

Cozzie et al. developed the Laika system[43] which uses Bayesian analysis to infer data structures from memory images. Pebbles differs from Laika in that it does not attempt to recover programmer defined data structures but to discover application-level data relationships from stable storage that would be recognizable and useful to an end user or developer.

Chapter 5

Conclusion and Future Work

5.1 Conclusions

I have argued that traditional data protection abstractions are insufficient to support emerging mobile and machine learning workloads and hypothesized that these new workloads have unique characteristics that we can leverage to build new data protection and management abstractions. As part of my research, I have developed three new protection systems that address exposure risks in machine learning workloads and mobile systems.

First, I described Sage, the first differentially private machine learning platform that enforces a global privacy guarantee limiting the amount of data exposed by globally deployed models. Sage contributes to new techniques to address challenges encountered while trying to deploy differential privacy. We developed block composition to support executing differentially private queries on an endless data stream We also developed privacy-adaptive training to balance the privacy-utility trade off while ensuring that all models released by Sage meet their configured quality goals.

Second, I describe Pyramid, a limited exposure data management framework. Pyramid's primary benefit is that it reduces that amount of data required to be accessed during model re-training. Pyramid leverage differentially private count tables to summarize historical data, allowing models to gain utility from the historical data without accessing it in its raw from. Pyramid's private count featurization enables more efficient learning and allow it to approach

the accuracy of state of the art classification models while training on two orders of magnitude less raw data.

Finally, I described Pebbles and its Logical Data Object (LDO) abstraction that allows data protection and management at a new level of granularity that would be impossible with traditional file system based management. Pebbles' design is guided by an extensive study of Android storage abstractions and leverages taint tracking and high level structured storage systems present in modern operating systems to build a device wide object graph to identify LDOs in unmodified mobile applications. Our evaluation showed that Pebbles' object identification was both accurate and efficient enough to build four new management applications.

5.2 Future Work

I think that the most impactful area of future work is not directly related to my research but in developing usable tooling for differential privacy. Those tools that do exist are mostly abandoned research project, are difficult to use, and are very easy to misuse. This echoes a statement from the U.S. Census Bureau which stated "there is a profound lack of toolkits for performing differential privacy calculations and for verifying the correctness of specific implementations"[66]. Despite it's theoretical maturity, I would argue that differential privacy is impossible for non-experts to use correctly in any but the most basic use cases.

Up to this point, most libraries for differential privacy have come with a statement in the documentation that saying the library "is NOT suitable for actual deployment at this point" which is included in the PINQ readme. Little effort has been put into developing libraries that are actually suitable for production deployment. There will be interesting research and development challenges that arise when attempting to harden a differentially privacy library for industrial use. Three important challenges that each build on the previous will be:

1. Validating differential privacy primitives.

2. Ensuring that all requirements are met to give a privacy guarantee.
3. Conveying to the developer what guarantee is given.

Validating the core primitives for differential privacy will be critical for any production library. This is akin to auditing cryptography libraries and will require a similar skillet. The most basic validation will be ensuring that the implementations of random distributions match mathematical definition. The guarantees of differential privacy are built upon real numbers but must be approximated in hardware using floating point values. Subtle nuances in floating point match can lead to breaking differentially private guarantees[115]. A production differential privacy library must ensure that all of the requirements are met to give a privacy guarantee. An example of the this challenge is that a machine learning library must sample from the entire dataset to implement the guarantee described in Abadi et al.[10]. On the surface this is simple but performant sampling is non-trivial for any large dataset. Finally, It must be very clear to the developer what guarantee a library is providing. Solving this challenge requires solving the first because seemingly small changes, such as changes in sampling, will effect the privacy guarantee.

I believe that meeting these challenges will require a different software design approach than existing machine learning and data processing systems. Most existing systems are built on loosely coupled operators that are mostly unaware of how the previous operators acted. The optimizer knows little about the input pipeline, which now little about how the data was preprocessed. Software for differential privacy will require very tightly coupled components where the overall system is acutely aware of how each component is operating on the data. I believe this is the only realistic design to achieve a usable library for differential privacy.

A new example of a differentially privacy library that makes strides in meeting these challenges is the recently released differentially private SQL library from Google[175;72]. This library implements a number of features important for a production deployment such as automatic bounds determination for setting the sensitivity of queries and the stochastic testing

protocol for ensuring that the library does not violate differential privacy. However, this library is not a full solution. It only gives a privacy guarantee on a per query basis and does not manage or account for privacy budget usage over time. There is still much work to be done in building a fully differentially private end-to-end system.

5.3 Final Thoughts

The goal of our work, and of differential privacy in general, is to improve the privacy of collected data and of data analysis. Sage, Pyramid, and other systems are built to make existing workflows more private and secure, and not to enable additional data collection. Organizations should not use these systems' stronger guarantees of security and privacy as a pretext to collect more data or to collect data that is otherwise considered too sensitive. Incentivizing additional data collection is not part of our research and is entirely antithetical to our goals.

Bibliography

[1] (2013). dm-crypt: Linux kernel device-mapper crypto target. https://code.google.com/p/cryptsetup/wiki/DMCrypt.

[2] (2014a). https://www.kaggle.com/c/criteo-display-ad-challenge/discussion/10429#54591.

[3] (2014b). Kaggle display advertising challenge dataset. https://www.kaggle.com/c/criteo-display-ad-challenge.

[4] (2015). Criteo releases its new dataset. http://labs.criteo.com/2015/03/criteo-releases-its-new-dataset/.

[5] (2015). Data lakes and the promise of unsiloed data. http://usblogs.pwc.com/emerging-technology/data-lakes-and-the-promise-of-unsiloed-data/.

[6] (2017). Federated learning: Collaborative machine learning without centralized training data. https://research.googleblog.com/2017/04/federated-learning-collaborative.html.

[7] (2018). Ai and compute. https://openai.com/blog/ai-and-compute/.

[8] (2018). Nyc taxi & limousine commission - trip record data. http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml.

[9] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. (2016a). Tensorflow: a system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283.

[10] Abadi, M., Chu, A., Goodfellow, I., McMahan, H. B., Mironov, I., Talwar, K., and Zhang, L. (2016b). Deep learning with differential privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 308–318. ACM.

[11] Agarwal, A., Bird, S., Cozowicz, M., Hoang, L., Langford, J., Lee, S., Li, J., Melamed, D., Oshri, G., Ribas, O., Sen, S., and Slivkins, A. (2016). A multiworld testing decision service. *CoRR*, abs/1606.03966.

[12] Agarwal, A., Hsu, D., Kale, S., Langford, J., Li, L., and Schapire, R. (2014). Taming the monster: A fast and simple algorithm for contextual bandits. In *Intl. Conf. on Machine Learning (ICML)*.

[13] Agarwal, P. K., Har-Peled, S., and Varadarajan, K. R. (2005). Geometric approximation via coresets. *Combinatorial and computational geometry*, 52:1–30.

[14] Agresti, A. (2013). *Categorical Data Analysis*. Wiley Series in Probability and Statistics. Wiley.

[15] Anand Basu (2010). Facebook Apps Leak User Information. http://www.reuters.com/article/2010/10/18/us-facebook-idUSTRE69H0QS20101018.

[16] Anderson, N. (2010). Why Google keeps your data forever, tracks you with ads. http://arstechnica.com/tech-policy/2010/03/google-keeps-your-data-to-learn-from-good-guys-fight-off-bad-guys/.

[17] Attariyan, M. and Flinn, J. (2010). Automating configuration troubleshooting with dynamic information flow analysis. In *Proc. of the Annual Network and Distributed System Security Symposium (NDSS)*.

[18] AzureML (2016). Build counting transform. https://msdn.microsoft.com/en-us/library/azure/mt243845.aspx.

[19] Backes, M., Berrang, P., Humbert, M., and Manoharan, P. (2016). Membership privacy in microRNA-based studies. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*.

[20] Baldi, P., Sadowski, P., and Whiteson, D. (2014). Searching for exotic particles in high-energy physics with deep learning. *Nature communications*, 5:4308.

[21] Barak, B., Chaudhuri, K., Dwork, C., Kale, S., McSherry, F., and Talwar, K. (2007). Privacy, accuracy, and consistency too: a holistic solution to contingency table release. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 273–282. ACM.

[22] Baylor, D., Breck, E., Cheng, H.-T., Fiedel, N., Foo, C. Y., Haque, Z., Haykal, S., Ispir, M., Jain, V., Koc, L., Koo, C. Y., Lew, L., Mewald, C., Modi, A. N., Polyzotis, N., Ramesh, S., Roy, S., Whang, S. E., Wicke, M., Wilkiewicz, J., Zhang, X., and Zinkevich, M. (2017). TFX: A Tensorflow-based production-scale machine learning platform. In *Proc. of the International Conference on Knowledge Discovery and Data Mining (KDD)*.

[23] Becker, A. (2015). Replacing Sawzall — a case study in domain-specific language migration. http://www.unofficialgoogledatascience.com/2015/12/replacing-sawzall-case-study-in-domain.html.

[24] Bernard, Tara Siegel and Hsu, Tiffany and Perlroth, Nicole and Lieber, Ron (2017). Equifax Says Cyberattack May Have Affected 143 Million in the U.S. https://www.nytimes.com/2017/09/07/business/equifax-cyberattack.html.

[25] Bilenko, M. (2016). Learning with counts. In preparation.

[26] Boyd, K., Lantz, E., and Page, D. (2015). Differential privacy for classifier evaluation. In *Proc. of the ACM Workshop on Artificial Intelligence and Security*.

[27] Brackett, M. (2012). *Data Resource Design: Reality Beyond Illusion*. IT Pro. Technics Publications Llc.

[28] Burges, C. J. (2010). *Dimension reduction: A guided tour*. Now Publishers Inc.

[29] Carlini, N., Liu, C., Erlingsson, U., Kos, J., and Song, D. (2018). The secret sharer: Evaluating and testing unintended memorization in neural networks. arXiv:1802.08232.

[30] Carlini, N., Liu, C., Erlingsson, Ú., Kos, J., and Song, D. (2019). The secret sharer: Evaluating and testing unintended memorization in neural networks. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 267–284.

[31] Chan, T.-H. H., Shi, E., and Song, D. (2011). Private and continual release of statistics. *ACM Transactions on Information Systems Security*.

[32] Chapelle, O., Manavoglu, E., and Rosales, R. (2014). Simple and scalable response prediction for display advertising. *ACM Trans. Intell. Syst. Technol.*, 5(4):61:1–61:34.

[33] Charikar, M., Chen, K., and Farach-Colton, M. (2002). Finding frequent items in data streams. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, ICALP '02, pages 693–703, London, UK, UK. Springer-Verlag.

[34] Chaudhuri, K. and Monteleoni, C. (2008). Privacy-preserving logistic regression. In *Proc. of the Conference on Neural Information Processing Systems (NeurIPS)*.

[35] Chaudhuri, K., Sarwate, A. D., and Sinha, K. (2013). A near-optimal algorithm for differentially-private principal components. *Journal of Machine Learning Research (JMLR)*, 14.

[36] Chen, Y., Pavlov, D., and Canny, J. F. (2009). Large-scale behavioral targeting. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, pages 209–218, New York, NY, USA. ACM.

[37] Chew, M. (2013). Writing for the 98%, blog post. `http://monica-at-mozilla.blogspot.com/2013/02/writing-for-98.html`.

[38] Chiu, O. (2015). Introducing Azure Data Lake. `https://azure.microsoft.com/en-us/blog/introducing-azure-data-lake/`.

[39] Chow, J., Pfaff, B., Garfinkel, T., Christopher, K., and Rosenblum, M. (2004). Understanding data lifetime via whole system simulation. In *Proc. of USENIX Security*.

[40] Chow, J., Pfaff, B., Garfinkel, T., and Rosenblum, M. (2005). Shredding your garbage: Reducing data lifetime through secure deallocation. In *Proc. of USENIX Security*.

[41] Cormode, G. and Muthukrishnan, S. (2005). An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75.

[42] Cox, L. P., Gilbert, P., Lawler, G., Pistol, V., Razeen, A., Wu, B., and Cheemalapati, S. (2014). Spandex: Secure password tracking for android. In *Proc. of USENIX Security*.

[43] Cozzie, A., Stratton, F., Xue, H., and King, S. T. (2008). Digging for data structures. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[44] Crankshaw, D., Bailis, P., Gonzalez, J. E., Li, H., Zhang, Z., Franklin, M. J., Ghodsi, A., and Jordan, M. I. (2014). The missing piece in complex analytics: Low latency, scalable model management and serving with Velox. *CoRR*, abs/1409.3809.

[45] Cummings, R., Krehbiel, S., Lai, K. A., and Tantipongpipat, U. (2018). Differential privacy for growing databases. In *Proc. of the Conference on Neural Information Processing Systems (NeurIPS)*.

[46] Dinur, I. and Nissim, K. (2003). Revealing information while preserving privacy. In *Proc. of the International Conference on Principles of Database Systems (PODS)*.

[47] Duchi, J. and Rogers, R. (2019). Lower bounds for locally private estimation via communication complexity. *arXiv:1902.00582*.

[48] Duchi, J. C., Jordan, M. I., and Wainwright, M. J. (2018). Minimax optimal procedures for locally private estimation. *Journal of the American Statistical Association*.

[49] Dudík, M., Langford, J., and Li, L. (2011). Doubly robust policy evaluation and learning. In *Intl. Conf. on Machine Learning (ICML)*, pages 1097–1104.

[50] Dunn, A. M., Lee, M. Z., Jana, S., Kim, S., Silberstein, M., Xu, Y., Shmatikov, V., and Witchel, E. (2012). Eternal sunshine of the spotless machine: Protecting privacy with ephemeral channels. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[51] Dwork, C. (2006). Differential privacy. In *Automata, languages and programming*, pages 1–12. Springer.

[52] Dwork, C. (2010). Differential privacy in new settings. In *Proceedings of the Twenty-first Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA.

[53] Dwork, C., McSherry, F., Nissim, K., and Smith, A. (2006). Calibrating noise to sensitivity in private data analysis. In *Proceedings of the Third Conference on Theory of Cryptography*, TCC'06, pages 265–284, Berlin, Heidelberg. Springer-Verlag.

[54] Dwork, C., Naor, M., Pitassi, T., , and Yekhanin, S. (2010a). Pan-private streaming algorithms. In *Proc. of The Symposium on Innovations in Computer Science*.

[55] Dwork, C., Naor, M., Pitassi, T., and Rothblum, G. N. (2010b). Differential privacy under continual observation. In *Proceedings of the forty-second ACM symposium on Theory of computing*.

[56] Dwork, C. and Roth, A. (2014). The algorithmic foundations of differential privacy. *Foundations and Trends® in Theoretical Computer Science*, 9(3–4):211–407.

[57] Dwork, C., Roth, A., et al. (2014). The algorithmic foundations of differential privacy. *Foundations and Trends® in Theoretical Computer Science*.

[58] Dwork, C., Rothblum, G. N., and Vadhan, S. (2010c). Boosting and differential privacy. In *Proc. of the IEEE Symposium on Foundations of Computer Science (FOCS)*.

[59] Dwork, C., Smith, A., Steinke, T., and Ullman, J. (2017). Exposed! A survey of attacks on private data. *Annual Review of Statistics and Its Application*.

[60] Dwork, C., Smith, A., Steinke, T., Ullman, J., and Vadhan, S. (2015). Robust traceability from trace amounts. In *Proc. of the IEEE Symposium on Foundations of Computer Science (FOCS)*.

[61] Ecular Xu (2017). Analyzing Xavier: An Information-Stealing Ad Library on Android. `https://blog.trendmicro.com/trendlabs-security-intelligence/analyzing-xavier-information-stealing-ad-library-android/`.

[62] Enck, W., Gilbert, P., Chun, B.-g., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N. (2010). TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[63] Eng, J. (2015). OPM hack: Government finally starts notifying 21.5 million victims. `http://www.nbcnews.com/tech/security/opm-hack-government-finally-starts-notifying-21-5-million-victims-n437126`.

[64] Feldman, D., Fiat, A., Kaplan, H., and Nissim, K. (2009). Private coresets. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 361–370. ACM.

[65] Fleischer, P. (2008). The European Commision's data protection findings. `http://googlepublicpolicy.blogspot.com/2008/04/european-commissions-data-protection.html`.

[66] Garfinkel, S. L., Abowd, J. M., and Powazek, S. (2018). Issues encountered deploying differential privacy. In *Proc. of the 2018 Workshop on Privacy in the Electronic Society (WPES'18)*.

[67] Geambasu, R., John, J. P., Gribble, S. D., Kohno, T., and Levy, H. M. (2011). Keypad: An auditing file system for theft-prone devices. In *Proc. of the ACM European Conference on Computer Systems (EuroSys)*.

[68] Gersho, A. and Gray, R. M. (2012). *Vector quantization and signal compression*, volume 159. Springer Science & Business Media.

[69] Goel, Vindu and Perlroth, Nicole (2016). Yahoo Says 1 Billion User Accounts Were Hacked. https://www.nytimes.com/2016/12/14/technology/yahoo-hack.html.

[70] Goodfellow, I., Bengio, Y., and Courville, A. (2016). Deep learning. Book in preparation for MIT Press.

[Google] Google. Storage options — android developers. http://developer.android.com/guide/topics/data/data-storage.html.

[72] Google (2019). Differential privacy. https://github.com/google/differential-privacy.

[73] Gorman, S. (2013). NSA officers spy on love interests. http://blogs.wsj.com/washwire/2013/08/23/nsa-officers-sometimes-spy-on-love-interests/.

[74] Gough, V. (2010). encfs. www.arg0.net/encfs.

[75] Graepel, T., Lauter, K., and Naehrig, M. (2012). Ml confidential: Machine learning on encrypted data. In *International Conference on Information Security and Cryptology*, pages 1–21. Springer.

[GRSecurity] GRSecurity. Homepage of pax. http://pax.grsecurity.net/.

[77] Gryta, T. (2015). T-Mobile customers' information compromised by data breach at credit agency. http://www.wsj.com/articles/experian-data-breach-may-have-compromised-roughly-15-million-consumers-1443732359.

[78] Gulshan, V., Peng, L., Coram, M., Stumpe, M. C., Wu, D., Narayanaswamy, A., Venugopalan, S., Widner, K., Madams, T., Cuadros, J., et al. (2016). Development and validation of a deep learning algorithm for detection of diabetic retinopathy in retinal fundus photographs. *Jama*, 316(22):2402–2410.

[79] Gutmann, P. (1996). Secure deletion of data from magnetic and solid-state memory. In *Proc. of USENIX Security*.

[80] Guyon, I. and Elisseeff, A. (2003). An introduction to variable and feature selection. *The Journal of Machine Learning Research*, 3:1157–1182.

[81] Halcrow, M. A. (2005). eCryptfs: An enterprise-class encrypted filesystem for linux. In *Proc. of the Linux Symposium*.

[82] Haldar, V., Chandra, D., and Franz, M. (2005). Dynamic taint propagation for java. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*.

[83] Harper, F. M. and Konstan, J. A. (2015). The MovieLens datasets: History and context. *ACM Trans. Interact. Intell. Syst.*, 5(4):19:1–19:19.

[84] Harter, T., Dragga, C., Vaughn, M., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. (2011). A file is not a file: Understanding the I/O behavior of Apple desktop applications. In *Proc. of the ACM Symposium on Operating Systems Principles (SOSP)*.

[85] Hazelwood, K., Bird, S., Brooks, D., Chintala, S., Diril, U., Dzhulgakov, D., Fawzy, M., Jia, B., Jia, Y., Kalro, A., Law, J., Lee, K., Lu, J., Noordhuis, P., Smelyanskiy, M., Xiong, L., and Wang, X. (2018). Applied machine learning at Facebook: A datacenter infrastructure perspective. In *Proc. of International Symposium on High-Performance Computer Architecture (HPCA)*.

[86] He, X., Pan, J., Jin, O., Xu, T., Liu, B., Xu, T., Shi, Y., Atallah, A., Herbrich, R., Bowers, S., et al. (2014). Practical lessons from predicting clicks on ads at facebook. In *Proceedings of the Eighth International Workshop on Data Mining for Online Advertising*, pages 1–9. ACM.

[87] Homer, N., Szelinger, S., Redman, M., Duggan, D., Tembe, W., Muehling, J., Pearson, J. V., Stephan, D. A., Nelson, S. F., and Craig, D. W. (2008). Resolving individuals contributing trace amounts of DNA to highly complex mixtures using high-density SNP genotyping microarrays. *PLoS Genetics*.

[88] Jayaraman, B. and Evans, D. (2019). Evaluating differentially private machine learning in practice. In *Proc. of USENIX Security*.

[89] Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., et al. (2017). In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 1–12.

[90] Kairouz, P., Oh, S., and Viswanath, P. (2015). The composition theorem for differential privacy. In *International Conference on Machine Learning (ICML)*.

[91] Kasiviswanathan, S. P., Lee, H. K., Nissim, K., Raskhodnikova, S., and Smith, A. (2011). What can we learn privately? *SIAM Journal on Computing*, 40(3):793–826.

[92] Kasiviswanathan, S. P. and Smith, A. (2014). On the'semantics' of differential privacy: A bayesian formulation. *Journal of Privacy and Confidentiality*.

[KeepSafe] KeepSafe. Hide pictures - KeepSafe Vault. `https://play.google.com/store/apps/details?id=com.kii.safe`.

[94] Kifer, D., Smith, A., and Thakurta, A. (2012). Private convex empirical risk minimization and high-dimensional regression. In *Proc. of the ACM Conference on Learning Theory (COLT)*.

[95] Koren, Y., Bell, R., and Volinsky, C. (2009). Matrix factorization techniques for recommender systems. *Computer*, (8):30–37.

[96] Langford, J., Li, L., and Strehl, A. (2007). Vowpal Wabbit online learning project.

[97] Langford, J. and Zhang, T. (2007). The Epoch-Greedy Algorithm for Contextual Multi-armed Bandits. In *Advances in Neural Information Processing Systems (NIPS)*.

[98] Lecuyer, M., Spahn, R., Vodrahalli, K., Geambasu, R., and Hsu, D. (2019). Privacy accounting and quality control in the sage differentially private ml platform. Online Supplements (also available on arXiv).

[99] Leonard, N. and Halasz, C. M. (2018). Twitter meets tensorflow. `https://blog.twitter.com/engineering/en_us/topics/insights/2018/twittertensorflow.html`.

[100] Li, L., Chu, W., Langford, J., and Schapire, R. E. (2010a). A contextual-bandit approach to personalized news article recommendation. In *Intl. World Wide Web Conf. (WWW)*.

[101] Li, L., Chu, W., Langford, J., and Wang, X. (2011). Unbiased offline evaluation of contextual-bandit-based news article recommendation algorithms. In *Intl. Conf. on Web Search and Data Mining (WSDM)*.

[102] Li, L. E., Chen, E., Hermann, J., Zhang, P., and Wang, L. (2017). Scaling machine learning as a service. In *Proceedings of The 3rd International Conference on Predictive Applications and APIs*, volume 67 of *Proceedings of Machine Learning Research*, pages 14–29, Microsoft NERD, Boston, USA. PMLR.

[103] Li, W., Wang, X., Zhang, R., Cui, Y., Mao, J., and Jin, R. (2010b). Exploitation and exploration in a performance based contextual advertising system. In Rao, B., Krishnapuram, B., Tomkins, A., and Yang, Q., editors, *KDD*, pages 27–36. ACM.

[104] Madden, M. and Smith, A. (2010). Reputation management and social media: How people monitor their identity and search for others online. `http://www.pewinternet.org/~/media/Files/Reports/2010/PIP_Reputation_Management_with_topline.pdf`.

[105] Mahoney, M. W. (2011). Randomized algorithms for matrices and data. *Foundations and Trends® in Machine Learning*, 3(2):123–224.

[106] McMahan, H. B. and Andrew, G. (2018a). A general approach to adding differential privacy to iterative training procedures. *arXiv:1812.06210*.

[107] McMahan, H. B. and Andrew, G. (2018b). A general approach to adding differential privacy to iterative training procedures. *arXiv:1812.06210*.

[108] McMahan, H. B., Holt, G., Sculley, D., Young, M., Ebner, D., Grady, J., Nie, L., Phillips, T., Davydov, E., Golovin, D., Chikkerur, S., Liu, D., Wattenberg, M., Hrafnkelsson, A. M., Boulos, T., and Kubica, J. (2013). Ad click prediction: a view from the trenches. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*.

[109] McMahan, H. B., Ramage, D., Talwar, K., and Zhang, L. (2018). Learning differentially private recurrent language models. In *Proc. of the International Conference on Learning Representations (ICLR)*.

[110] McSherry, F. and Mironov, I. (2009). Differentially private recommender systems: Building privacy into the Netflix prize contenders. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 627–636. ACM.

[111] McSherry, F. D. (2009). Privacy integrated queries: An extensible platform for privacy-preserving data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 19–30, New York, NY, USA. ACM.

[112] Melis, L., Danezis, G., and De Cristofaro, E. (2016). Efficient private statistics with succinct sketches. In *Network and Distributed System Security Symposium–NDSS 2016*.

[113] Microsoft Corporation (2009). Windows 7 BitLocker executive overview. `http://technet.microsoft.com/en-us/library/dd548341(WS.10).aspx`.

[114] Mir, D., Muthukrishnan, S., Nikolov, A., and Wright, R. N. (2011). Pan-private algorithms via statistics on sketches. In *Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 37–48. ACM.

[115] Mironov, I. (2012). On significance of the least significant bits for differential privacy. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 650–661. ACM.

[116] Mohan, P., Thakurta, A., Shi, E., Song, D., and Culler, D. (2012). GUPT: Privacy preserving data analysis made easy. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 349–360. ACM.

[117] Muniswamy-Reddy, K.-K., Holland, D. A., Braun, U., and Seltzer, M. (2006). Provenance-aware storage systems. In *Proc. of the USENIX Annual Technical Conference (ATC)*.

[118] Muniswamy-Reddy, K.-K., Macko, P., and Seltzer, M. (2010). Provenance for the cloud. In *Proc. of the USENIX Conference on File and Storage Technologies (FAST)*.

[119] Muthukrishnan, S. (2005). *Data streams: Algorithms and applications*. Now Publishers Inc.

[120] Narayanan, A. and Shmatikov, V. (2008). Robust de-anonymization of large sparse datasets. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, SP '08, pages 111–125, Washington, DC, USA. IEEE Computer Society.

[121] Newsome, J. and Song, D. (2005). Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. of the Annual Network and Distributed System Security Symposium (NDSS)*.

[122] Nikolaenko, V., Weinsberg, U., Ioannidis, S., Joye, M., Boneh, D., and Taft, N. (2013). Privacy-preserving ridge regression on hundreds of millions of records. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 334–348. IEEE.

[123] Nissim, K., Raskhodnikova, S., and Smith, A. (2007). Smooth sensitivity and sampling in private data analysis. In *Proceedings of the Thirty-ninth Annual ACM Symposium on Theory of Computing*, STOC '07, pages 75–84, New York, NY, USA. ACM.

[NQ Mobile Security] NQ Mobile Security. Vault-Hide SMS, Pics & Videos. `https://play.google.com/store/apps/details?id=com.netqin.ps`.

[125] Ornstein, C. (2015). Celebrities' medical records tempt hospital workers to snoop. `https://www.propublica.org/article/clooney-to-kardashian-celebrities-medical-records-hospital-workers-snoop`.

[126] Pappas, V., Kemerlis, V. P., Zavou, A., Polychronakis, M., and Keromytis, A. D. (2013). CloudFence: Data flow tracking as a cloud service. In *Proc. of the Symposium on Research in Attacks, Intrusions and Defenses*.

[127] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

[Pendragon Software Corporation] Pendragon Software Corporation. Caffeinemark 3.0. `http://www.benchmarkhq.ru/cm30/`.

[129] Perlman, R. (2005). File system design with assured delete. In *Proc. of the IEEE International Security in Storage Workshop (SISW)*.

[130] Popa, R. A., Redfield, C., Zeldovich, N., and Balakrishnan, H. (2011). CryptDB: Protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 85–100. ACM.

[131] Proserpio, D., Goldberg, S., and McSherry, F. (2014). Calibrating data to sensitivity in private data analysis: a platform for differentially-private analysis of weighted datasets. *Proc. of the International Conference on Very Large Data Bases (VLDB)*.

[132] Rao, L. (2012). Google consolidates privacy policy; will combine user data across services. `http://techcrunch.com/2012/01/24/google-consolidates-privacy-policy-will-combine-user-data-across-services/`.

[133] Ravi, S. (2017). On-device machine intelligence. `https://ai.googleblog.com/2017/02/on-device-machine-intelligence.html`.

[134] Reardon, J., Capkun, S., and Basin, D. (2012). Data node encrypted file system: Efficient secure deletion for flash memory. In *Proc. of USENIX Security*.

[135] Rogers, R., Roth, A., Ullman, J., and Vadhan, S. (2018). Privacy odometers and filters: Pay-as-you-go composition. In *Proc. of the Conference on Neural Information Processing Systems (NeurIPS)*.

[136] Rogers, R. M., Roth, A., Ullman, J., and Vadhan, S. (2016). Privacy odometers and filters: Pay-as-you-go composition. In *Advances in Neural Information Processing Systems*.

[137] Rostin, A., Albrecht, O., Bauckmann, J., Naumann, F., and Leser, U. (2009). A machine learning approach to foreign key discovery. In *Proc. of the International Workshop on the Web and Databases (WebDB)*.

[138] Roy, I., Porter, D. E., Bond, M. D., McKinley, K. S., and Witchel, E. (2009). Laminar: practical fine-grained decentralized information flow control. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[139] Roy, I., Setty, S. T., Kilzer, A., Shmatikov, V., and Witchel, E. (2010). Airavat: Security and privacy for MapReduce. In *NSDI*, volume 10, pages 297–312.

[140] Russell, S. J. and Norvig, P. (2003). *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition.

[141] Saltzer, J. H. (1974). Protection and the control of information sharing in Multics. *Communications of the ACM*.

[142] Sanger, David E. and Perlroth, Nicole and Thrush, Glenn and Rappeport, Alan (2018). Marriott Data Breach Is Traced to Chinese Hackers as U.S. Readies Crackdown on Beijing. `https://www.nytimes.com/2018/12/11/us/politics/trump-china-trade.html`.

[143] Schneier, B. (2015). Data is a toxic asset. `https://www.schneier.com/blog/archives/2016/03/data_is_a_toxic.html`.

[SEAndroid] SEAndroid. SEforAndroid. `http://selinuxproject.org/page/SEAndroid`.

[SELinux] SELinux. Selinux project wiki. `http://selinuxproject.org/page/Main_Page`.

[Seltzer] Seltzer, M. Pass: Provenance-aware storage systems. `http://www.eecs.harvard.edu/syrah/pass/`.

[147] Sen, S., Guha, S., Datta, A., Rajamani, S. K., Tsai, J., and Wing, J. M. (2014). Bootstrapping privacy compliance in big data systems. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*.

[148] Seth Weintraub (2011). Industry first: Smartphones pass PCs in sales. `https://fortune.com/2011/02/07/industry-first-smartphones-pass-pcs-in-sales/`.

[149] Settles, B. (2012). Active learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 6(1):1–114.

[150] Shalev-Shwartz, S. (2011). Online learning and online convex optimization. *Foundations and Trends in Machine Learning*, 4(2):107–194.

[151] Shalev-Shwartz, S. and Ben-David, S. (2014). *Understanding Machine Learning: From Theory to Algorithms. Appendix B.* Cambridge University Press, New York, NY, USA.

[152] Shi, Q., Petterson, J., Dror, G., Langford, J., Smola, A., and Vishwanathan, S. (2009). Hash kernels for structured data. *The Journal of Machine Learning Research*, 10:2615–2637.

[153] Shiebler, D. and Tayal, A. (2010). Making machine learning easy with embeddings. SysML `http://www.sysml.cc/doc/115.pdf`.

[154] Shokri, R., Stronati, M., Song, C., and Shmatikov, V. (2017). Membership inference attacks against machine learning models. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 3–18. IEEE.

[155] Shrivastava, A., Konig, A. C., and Bilenko, M. (2016). Time adaptive sketches (ada-sketches) for summarizing data streams. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1417–1432. ACM.

[156] Smith, A. and Thakurta, A. (2013). Differentially private model selection via stability arguments and the robustness of lasso. *Journal of Machine Learning Research*.

[157] Soules, C. A. and Ganger, G. R. (2005). Connections: using context to enhance file search. In *Proc. of the ACM Symposium on Operating Systems Principles (SOSP)*.

[158] Srivastava, A., König, A. C., and Bilenko, M. (2016). Time adaptive sketches (ada-sketches) for summarizing data streams. In *ACM SIGMOD Conference*. ACM.

[159] Strimel, G. P., Sathyendra, K. M., and Peshterliev, S. (2018). Statistical model compression for small-footprint natural language understanding. arXiv:1807.07520.

[160] Symantec Corporation (2012). PGP whole disk encryption. `http://www.symantec.com/whole-disk-encryption`.

[161] Talwar, K., Thakurta, A., and Zhang, L. (2015). Nearly-optimal private LASSO. In *Proc. of the Conference on Neural Information Processing Systems (NeurIPS)*.

[162] Tang, Y., Ames, P., Bhamidipati, S., Bijlani, A., Geambasu, R., and Sarda, N. (2012). CleanOS: Mobile OS abstractions for managing sensitive data. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[163] Tang, Y., Lee, P. P., Lui, J. C., and Perlman, R. (2010). FADE: Secure overlay cloud storage with file assured deletion. In *Proc. of the International ICST Conference on Security and Privacy in Communication Networks (SecureComm)*.

[164] The Chaos Computing Club (CCC) (2013). CCC breaks Apple TouchID. `http://www.ccc.de/en/updates/2013/ccc-breaks-apple-touchid`.

[165] Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 267–288.

[166] Tramèr, F., Zhang, F., Juels, A., Reiter, M. K., and Ristenpart, T. (2016). Stealing machine learning models via prediction apis. In *Proc. of USENIX Security*.

[167] TrueCrypt Foundation (2007). Truecrypt – free open-source on-the-fly encryption. `http://www.truecrypt.org/`.

[168] Tu, S., Kaashoek, M. F., Madden, S., and Zeldovich, N. (2013). Processing analytical queries over encrypted data. In *Proceedings of the VLDB Endowment*. VLDB Endowment.

[169] Valentin-DeVries, J., Singer, N., Keller, M., and Krolik, A. (2018). Your apps know where you were last night, and they're not keeping it secret. https://www.nytimes.com/interactive/2018/12/10/business/location-data-privacy-apps.html.

[170] Vandebogart, S., Efstathopoulos, P., Kohler, E., Krohn, M., Frey, C., Ziegler, D., Kaashoek, F., Morris, R., and Mazières, D. (2007). Labels and event processes in the Asbestos operating system. *ACM Transactions on Computer Systems (TOCS)*.

[171] Viennot, N., Garcia, E., and Nieh, J. (2014). A measurement study of google play. In *Proc. of the ACM International Conference on Measurement and Modeling of Computer Systems*.

[172] Wang, Y.-X. (2018). Revisiting differentially private linear regression: optimal and adaptive prediction & estimation in unbounded domain. *arXiv preprint arXiv:1803.02596*.

[173] Weinberger, K., Dasgupta, A., Langford, J., Smola, A., and Attenberg, J. (2009). Feature hashing for large scale multitask learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 1113–1120. ACM.

[174] Wilson, D. (2014). Hearst's VP of data on connecting the data dots. `http://www.pubexec.com/article/hearsts-vp-data-connecting-data-dots/`.

[175] Wilson, R., Zhang, C. Y., Lam, W., Desfontaines, D., Simmons-Marengo, D., and Gipson, B. (2019). Differentially private sql with bounded user contribution. *arXiv preprint arXiv:1909.01917*.

[176] Wu, C.-J., Brooks, D., Chen, K., Chen, D., Choudhury, S., Dukhan, M., Hazelwood, K., Isaac, E., Jia, Y., Jia, B., Leyvand, T., Lu, H., Lu, Y., Qiao, L., Reagen, B., Spisak, J., Sun, F., Tulloch, A., Vajda, P., Wang, X., Wang, Y., Wasti, B., Wu, Y., Xian, R., Yoo, S., and Zhang, P. (2019). Machine learning at Facebook: Understanding inference at the edge. In *Proc. of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*.

[177] Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., Klingner, J., Shah, A., Johnson, M., Liu, X., Kaiser, L., Gouws, S., Kato, Y., Kudo, T., Kazawa, H., Stevens, K., Kurian, G., Patil, N., Wang, W., Young, C., Smith, J., Riesa, J., Rudnick, A., Vinyals, O., Corrado, G., Hughes, M., and Dean, J. (2016). Google's neural machine translation system: Bridging the gap between human and machine translation. *CoRR*, abs/1609.08144.

[178] Wulf, W. A., Cohen, E. S., Corwin, W. M., Jones, A. K., Levin, R., Pierson, C., and Pollack, F. J. (1974). Hydra: The kernel of a multiprocessor operating system. *Communications of the ACM*.

[179] Xiao, X., Bender, G., Hay, M., and Gehrke, J. (2011). iReduct: Differential privacy with reduced relative errors. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 229–240. ACM.

[180] Xu, J., Zhang, Z., Xiao, X., Yang, Y., Yu, G., and Winslett, M. (2012). Differentially private histogram publication. In *Proc. of the IEEE International Conference on Data Engineering (ICDE)*.

[181] Yip, A., Wang, X., Zeldovich, N., and Kaashoek, M. F. (2009). Improving application security with data flow assertions. In *Proc. of the ACM Symposium on Operating Systems Principles (SOSP)*.

[182] Yu, L., Liu, L., Pu, C., Gursoy, M. E., and Truex, S. (2019). Differentially private model publishing for deep learning. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*.

[183] Zhang, D., McKenna, R., Kotsogiannis, I., Hay, M., Machanavajjhala, A., and Miklau, G. (2018). Ektelo: A framework for defining differentially-private computations. In *Proc. of the ACM SIGMOD International Conference on Management of Data*.

[184] Zhang, J., Zhang, Z., Xiao, X., Yang, Y., and Winslett, M. (2012). Functional mechanism: Regression analysis under differential privacy. In *Proc. of the International Conference on Very Large Data Bases (VLDB)*.

[185] Zhang, M., Hadjieleftheriou, M., Ooi, B. C., Procopiuc, C. M., and Srivastava, D. (2010). On multi-column foreign key discovery. *Proceedings of the VLDB Endowment*.

[186] Zhang, Y., Yang, M., Xu, B., Yang, Z., Gu, G., Ning, P., Wang, X., and Zang, B. (2013). Vetting undesirable behaviors in android apps with permission use analysis. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*.

[187] Zhu, D. Y., Jung, J., Song, D., Kohno, T., and Wetherall, D. (2011). TaintEraser: protecting sensitive data leaks using application-level taint tracking. *ACM SIGOPS Operating Systems Review*.

[188] Zhu, X. (2006). Semi-supervised learning literature survey.